ST200 VLIW Series

ST200 Micro Toolset User Manual

Last updated 15 June 2005

Issued by the MCDT Documentation Group on behalf of STMicroelectronics

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without the express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics.
All other names are the property of their respective owners.

The ST200 series cores are based on technology jointly developed by Hewlett-Packard Laboratories and STMicroelectronics.

Microsoft® and Windows® are registered trademarks of Microsoft Corporation in the United States and/or other countries. Solaris™ is a trademark of Sun Microsystems, Inc. in the US and other countries. Cygwin™ and Insight™ are trademarks of Red Hat Software, Inc. Linux® is a registered trademark of Linus Torvalds.

© 2002, 2003, 2004, 2005 STMicroelectronics. All Rights Reserved.

STMicroelectronics Group of Companies
Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States.

http://www.st.com

Contents

Preface **xiii**

ST200 document identification and control xiii
ST200 documentation suite xiii
Conventions used in this guide xv

1 ST200 development system **1**

1.1 ST200 compiler overview 1
1.2 ST200 C++ compiler 2
1.3 Toolset overview 3
1.4 ST200 Micro Toolset interface 5
1.4.1 Example command-lines 6

2 st200cc **9**

2.1 Getting started 9
2.1.1 Invoking the compiler 9
2.1.2 Input and output 10
2.2 Command-line options 11
2.2.1 Getting help 11
2.2.2 Overall options 11
2.2.3 C preprocessor options 12

2.2.4 C dialect options 13
2.2.5 Warning options 13
2.2.6 Debugging options 16
2.2.7 Profiling options 16
2.2.8 Call Trace instrumentation options 16
2.2.9 Optimization options 16
2.2.10 Code generation options 17
2.2.11 -OPT options 19
2.2.12 -Data prefetching options 20
2.2.13 Inlining options 20
2.2.14 Interprocedural analysis 20
2.2.15 Symbol visibility specification 20
2.2.16 Sending options to a specific phase 21
2.2.17 Specifying the run-time environment 21
2.2.18 Selecting the endianness 22
2.2.19 Configuration options 22
2.2.20 Directory and library options 23
2.2.21 Environment variables 24
2.3 Predefined macros 24
2.4 C99 support 27

3 st200c++ 31

3.1 Introduction to st200c++ 31
3.1.1 References 31
3.2 Getting started 32
3.2.1 Invoking the compiler 32
3.2.2 Input and output 32
3.3 C++ specific command-line options 33
3.3.1 C++ dialect options 33
3.3.2 C++ specific warning options 33
3.3.3 Code generation options 34
3.3.4 Directory and library options 36

3.4 Predefined macros 36
3.5 GNU C++ language extensions 38
3.5.1 Extensions to the C++ language 38
3.5.2 Variable, functions and type attributes 38
3.5.3 Built-ins 39
3.6 GNU C++ asm statement 39
3.7 Standard Template Libraries (STL) support 39
3.7.1 Standard headers status 40
3.7.2 Supported extension headers: 41
3.7.3 Supported backward compatibility headers: 42
3.7.4 Identifying the STL version 43
3.8 RTTI notes 43
3.9 Exceptions notes 44
3.10 Checking operator new return value 44
3.11 Known limitations 46
3.11.1 Status of the known limitations 46

4 Pragmas 47

4.1 Pragmas short description and syntax 47
4.2 Loop optimization pragmas 49
4.2.1 #pragma unroll 49
4.2.2 #pragma ivdep 50
4.2.3 #pragma loopdep 51
4.2.4 #pragma loopmod 53
4.2.5 #pragma looptrip 54
4.2.6 #pragma pipeline 55
4.2.7 Code generation pragmas 56
4.2.8 Heuristic pragmas 57
4.3 Miscellaneous pragmas 58

4.3.1 #pragma ident 58
4.3.2 #pragma weak 58

5 Optimization guide 59

5.1 Introduction 59
5.2 Inlining 59
5.2.1 Single file inlining 59
5.2.2 st200cc inlining options 61
5.2.3 Extern inline functions 63
5.2.4 Inlining pragmas 63
5.3 Memory dependences in C programs 69
5.4 Aliasing rules in C/C++ programs 71
5.5 Data prefetching 75
5.5.1 Manual data prefetching 76
5.5.2 Automatic data prefetching 76
5.5.3 Advanced data prefetching options 79
5.6 Profiling 80
5.7 Call trace instrumentation 81
5.7.1 Instrumenting functions: -finstrument-functions 81
5.7.2 Instrumenting call to functions: -minstrument-calls 81
5.8 Profiling feedback optimization (PFO) 83
5.8.1 Principles 83
5.8.2 Command line 83
5.8.3 Example 84
5.8.4 What feedback does 85
5.9 Interprocedural analysis optimization (IPA) 86
5.9.1 Introduction 86
5.9.2 Using IPA 86
5.9.3 IPA command line options 87
5.9.4 Limitations 88
5.10 Symbol visibility specification 89

5.10.1 Introduction 89
5.10.2 Usage of the -fvisibility option 91
5.10.3 Usage of the -mvisibility-decl option 92
5.10.4 The visibility specification file 94

6 GNU C extensions supported by st200cc 97

6.1 Introduction 97
6.2 Extensions to the C language family 98
6.2.1 Statements and declarations in expressions 98
6.2.2 Locally declared labels 99
6.2.3 Labels as values 99
6.2.4 Naming an expression's type 100
6.2.5 Referring to a type with typeof 100
6.2.6 Generalized L-values 100
6.2.7 Conditionals with omitted operands 101
6.2.8 Double-word integers 101
6.2.9 Hexadecimal floats 101
6.2.10 Specifying a register for a local variable 101
6.2.11 Array of length zero 102
6.2.12 Array of variable length 103
6.2.13 Macro with variable number of arguments 104
6.2.14 Strings literals with embedded newlines 105
6.2.15 Non-Lvalue arrays may have subscripts 106
6.2.16 Arithmetic on void and function pointers 106
6.2.17 Non-constant initializers 107
6.2.18 Compound literals 107
6.2.19 Designated initializers 108
6.2.20 Case ranges 108
6.2.21 Cast to a union type 109
6.2.22 Dollar signs in identifier names 109
6.2.23 Prototypes and old-style function definitions 109
6.2.24 C++ comments 109
6.2.25 Character ESC in constants 110

6.2.26 Inquiring on alignment of types or variables 110
6.2.27 Incomplete enum type 110
6.2.28 Function names as strings 110
6.3 Attributes 111
6.3.1 Placement and layout 111
6.3.2 Optimization 114
6.3.3 Visibility attributes 114
6.3.4 Miscellaneous attributes 115
6.3.5 Built-ins 116

7 GNU ASM 119

7.1 Introduction 119
7.1.1 Syntax 119
7.1.2 Assumptions 120
7.1.3 Volatile 121
7.1.4 Scheduling considerations 121
7.1.5 Restrictions 122
7.1.6 Example 122

8 Intrinsic functions 123

8.1 Introduction 123
8.1.1 Rationale 123
8.1.2 Models 123
8.2 Naming intrinsics 124
8.2.1 Naming scheme 124
8.2.2 Building intrinsic names 125
8.3 Using intrinsics from C/C++ 130
8.3.1 Include files 130
8.4 Understanding intrinsic models 133
8.4.1 Understanding fundamentals operators 133
8.4.2 Understanding models 134

8.5 Intrinsic functions summary 134
8.5.1 Functions 135
8.6 ST231 intrinsics 145
8.7 Division and modulus built-ins 146

9 Compiler bugs 147

9.1 Introduction 147
9.2 Identifying a compiler bug 147
9.2.1 Category 1 147
9.2.2 Category 2 148
9.3 Checks performed by user 148
9.4 Work-around 148
9.5 Reporting a compiler bug 149
9.6 Known bugs and limitations 149

10 ICache/dead code optimization 151

10.1 Introduction 151
10.1.1 How it works 152
10.1.2 Synopsis 153
10.1.3 ICache files 154
10.2 st200cc options 156
10.2.1 Other options 160
10.2.2 Default behavior 160
10.2.3 Option combinations 161
10.2.4 Full optimization 161
10.2.5 Relocatable files 161
10.2.6 Shared libraries, call shared executable 162
10.2.7 Passing other options to the optimization phase 162
10.3 Examples 163
10.3.1 Compiling with static ICache optimizations 163

10.3.2 Compiling a relocatable object with static ICache optimizations 164
10.3.3 To keep a generated layout file 164
10.3.4 Using profile driven ICache optimizations 164
10.3.5 Using profile driven ICache optimizations using -O2 or higher 165
10.3.6 Compiling with st200gprof file-driven ICache optimizations 165
10.3.7 Linker options 166
10.4 binopt 166
10.4.1 Synopsis 166
10.4.2 Options 167
10.4.3 binopt dump options 170
10.4.4 How to configure binopt 171
10.4.5 User command file 172
10.5 Warning and error messages 172
10.5.1 binopt 172
10.6 References 174

11 Assembler 175

11.1 Introduction 175
11.2 Assembler syntax 176
11.2.1 General description 176
11.2.2 Comments 176
11.2.3 Lexical categories 177
11.2.4 Symbols 179
11.2.5 Expressions 181
11.2.6 Bundles 182
11.2.7 Operands 183
11.2.8 Operations 185
11.2.9 Assembler directives 187
11.3 Invoking the assembler 192

11.3.1 Assembler command line 192
11.3.2 Error and warning messages 194

Revision history 195

Index 197

Preface

This document is part of the ST200 documentation suite detailed below. Comments on this or other manuals in the ST200 documentation suite should be made by contacting your local STMicroelectronics Limited sales office or distributor.

ST200 document identification and control

Each book in the ST200 documentation suite carries a unique ADCS identifier in the form:

ADCS *nnnnnnnx*

Where, *nnnnnnn* is the document number and *x* is the revision.

Whenever making comments on a ST200 document the complete identification ADCS *nnnnnnnx* should be quoted.

ST200 documentation suite

The ST200 documentation suite comprises the following volumes:

ST200 Micro Toolset User Manual

(ADCS 7508723) This manual describes the software provided as part of the ST200 tools. It supports the development of ST200 applications for embedded systems. Applications may be developed in either a stand-alone environment, or under the OS21 real-time operating system.

This manual also contains reference material relating to the ST200 Micro Toolset.

ST200 Cross Development Manual

(ADCS 7521642) This manual describes the cross development tools and platforms.

ST200 Run-time Architecture Manual

(ADCS 7521848) This manual describes the common software conventions for the ST200 processor runtime architecture.

OS21 for ST200 User Manual

(ADCS 7410372) This manual describes the use of OS21 on ST200 platforms. It describes how specific ST200 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST200 platforms.

ST220 Core and Instruction Set Architecture

(ADCS 7395369) This manual describes the architecture and the instruction set of the ST220 core as used by STMicroelectronics.

ST231 Core and Instruction Set Architecture

(ADCS 7645929) This manual describes the architecture and the instruction set of the ST231 core as used by STMicroelectronics.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- **`sample code`**, **`keyboard input`** and **`file names`**,
- *variables* and ***`code variables`***,
- `code comments`,
- **screens**, **windows** and **dialog boxes**,
- **instructions**.

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,
- PIN NAMES and SIGNAL NAMES.

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1 Terminal strings of the language, that is, those not built up by rules of the language, are printed in teletype font. For example, **`void`**.

2 Nonterminal strings of the language, that is, those built up by rules of the language, are printed in italic teletype font. For example, ***`name`***.

3 If a nonterminal string of the language starts with a non-italicized part, it is equivalent to the same non-terminal string without that non-italicized part. For example, **`vspace-`*`name`***.

4 Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘**`::=`**’).

5 Alternatives are separated by vertical bars (‘**`|`**’).

6 Optional sequences are enclosed in square brackets (‘**`[`**’ and ‘**`]`**’).

7 Items which may be repeated appear in braces (‘**`{`**’ and ‘**`}`**’).

1 ST200 development system

1.1 ST200 compiler overview

The purpose of the **st200cc** compilation driver is to translate a program written in the C language, into the ST220 assembly language so that is suitable for assembly, linking, and execution. The assembler file is compiled using **st200as** and linked using **st200ld** to provide an ST200 binary image. All these phases are hidden using the driver tool **st200cc**.

The **st200cc** compiler uses the GNU C language parser, and implements state-of-the art compiler optimizations. Thanks to this GNU C language parser, the **st200cc** compiler is closely compatible with the GNU C compiler, both at the driver level, and on C language extensions (GNU Compiler Collection project http://www.gnu.org/software/gcc/gcc.html). The processor-independent compiler optimizations available in the **st200cc** compiler are mostly inherited from the Open64 project hosted on SourceForge http://open64.sourceforge.net. Other compiler optimizations that are specific to the ST200 family of processors were developed by STMicroelectronics at the CMG Compilation and Simulation Expertise center.

These include:

- the exploitation of the ST200 SELECT instruction,
- aggressive instruction selection including mapping of the user boolean variables to the branch registers,
- instruction scheduling,
- software pipelining of the inner loops,
- compiler intrinsics and builtins support,

- reordering of object code to minimize instruction cache conflicts,
- dead code and dead data elimination,
- automatic generation of prefetch instructions,
- profiling feedback optimization,
- interprocedural analysis optimization.

The binary image can be executed on an ST220, or ST231 hardware target or by using **st200gdb**. The binary format used for the image is ELF and the debug format is DWARF2. The ST200 Micro Toolset supports both little endian and big endian code generation for ST200 targets.

Where applicable, the available options are accessible through a command-line interface similar to the UNIX style. This will be familiar to most **gcc** and **cc** users. The toolset is installed in a directory structure which also follows the UNIX structure, that is **`bin`** and **`lib`**. This is described in more detail in *Section 1.4: ST200 Micro Toolset interface on page 5*.

The compiler supports the ANSI C89 standard and partially supports the ANSI C99 standard, see *Section 2.4: C99 support on page 27*.

1.2 ST200 C++ compiler

A GNU C++ compiler is provided which is documented in *Chapter 3: st200c++ on page 31*. Command line options which are common to both **st200cc** and **st200c++** are documented in *Chapter 2: st200cc on page 9*, so C++ users should read both chapters.

1.3 Toolset overview

The ST200 Micro Toolset is a set of tools that allow C and C++ programs compiled for an ST200 target to be simulated on a host workstation or executed on an ST200 target. Supported platforms are:

- Solaris 2.5.8,
- RedHat Linux 7.2,
- RedHat Enterprise Linux 3.0,
- Windows XP with any Cygwin environment from version 1.5.11 or higher,
- Windows 2000 with any Cygwin environment from version 1.5.11 or higher.

The ST200 Micro Toolset is mainly intended for tool developers, for operating system development and for applications that require modeling interrupts and real-time behavior. It includes the whole set of tools that manipulate ST200 object files, including the ST200 assembler, compiler, linker, load/run tool, debugger and archiver. Here, ST200 assembler files are translated to ST200 object files that the linker merges to produce an ST200 executable image. This image file does not run natively on the host workstation and requires an **interpreter** to be executed. See *Section 1.4.1: Example command-lines on page 6* for details.

Figure 1 shows the main components of the ST200 Micro Toolset.

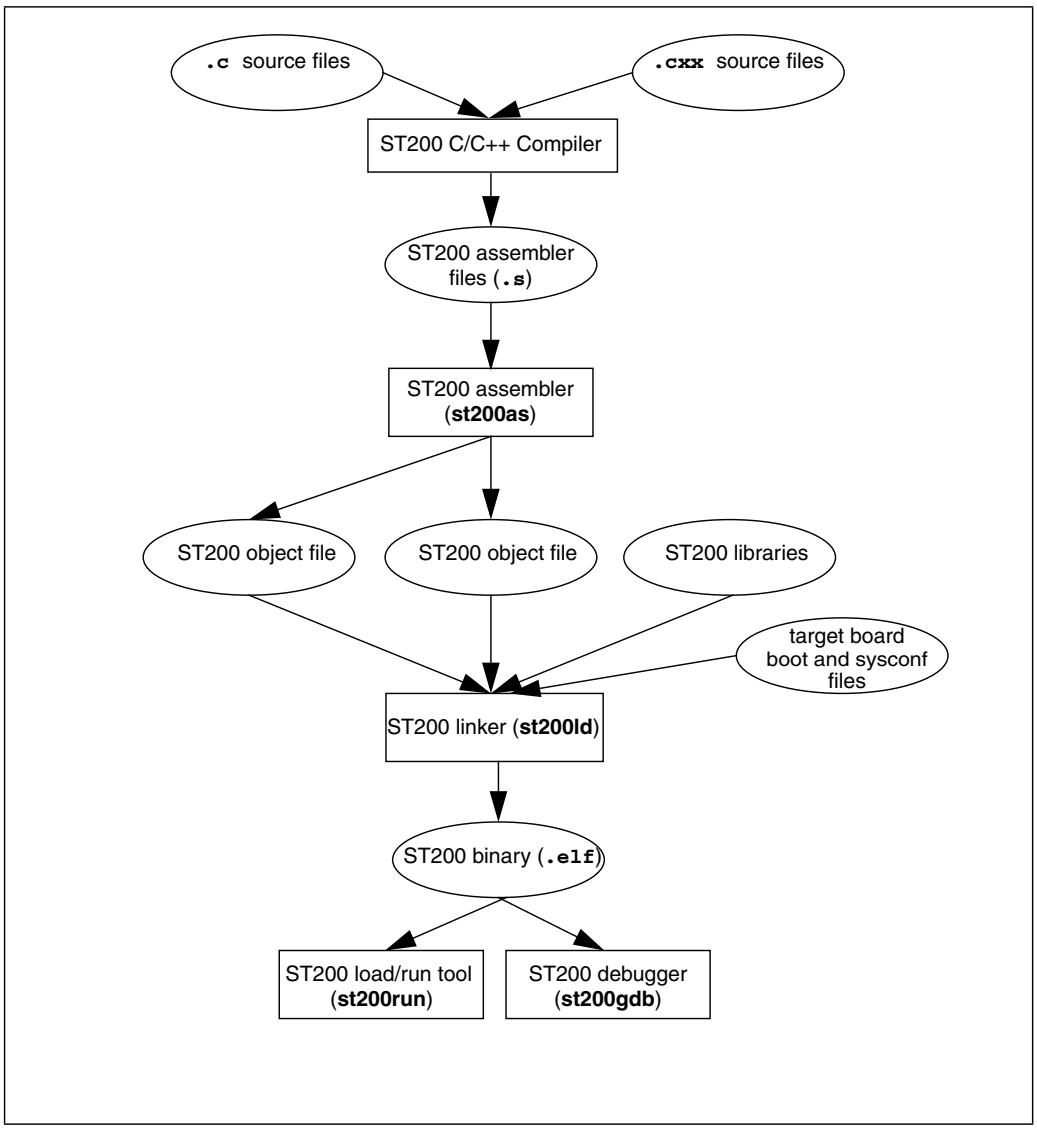

Figure 1: Components of the ST200 Micro Toolset interfaces

Note: Figure 1 does not include the binary optimizer tool (instruction cache placement, dead code and dead data elimination) nor the PFO (Profiling Feedback Optimization) nor the IPA (Interprocedural Analysis Optimization).

1.4 ST200 Micro Toolset interface

The interface to the toolset is a **command-line interface** and makes the assumption that a **command-line shell** (like **sh**, **ksh**, **csh**, **tcsh**, or similar) is available to the user. On Windows XP/2000 systems, this is provided by the CYGWIN32 system (see http://sources.redhat.com/cygwin for details). Most of the ST200 tools work as command-line interfaces, with the exception of the debugger (**st200gdb**) that is optionally available with a **Tcl/Tk** interface.

The steps involved in getting from a C program to an ST200 executable image are hidden from the user and implemented by the **st200cc** driver. The ST200 Micro Toolset supports the ST220 and ST231 cores, both little endian and big endian code generation and both bare machine and OS21 run-time environments. It is installed in a directory tree with the structure shown below. (The example shows part of a Windows directory tree.)

***<tools-dir>*/bin**	**st200ar.exe**	(archiver)
	st200as.exe	(assembler)
	st200cc.exe	(C compiler driver)
	st200c++.exe	(C++ compiler driver)
	st200gdb.exe	(debugger)
	st200ld.exe	(linker)
	st200run.exe	driver for simulator or hardware
	<other binary utilities>	
***<tools-dir>*/lib/cmplrs**	<all the compiler components>	
<tools-dir>*/lib/*<core_target>*/ *<endianess>*/*<run-time>	**crt*xxx*.o**	C runtime start-up files
	libgcc.a	libraries
	libgloss.a	
	libgprof.a	
	libc.a	
	libm.a	

<*tools-dir*>/lib/engine/<*core_target*>/<*endianess*>	**adaptor.rcu** **basicsim.dll** **st200emu.dll** **gdiserv.exe** **gdiserv.dll** **stmrpc.dll** **st200sim.dll**	execution engines
<*tools-dir*>/include	<all the include files>	
<*tools-dir*>/*target*		the target tree containing all of the BSP files relative to the core, board and soc, see the *ST200 Cross Development Manual* for more information.
	targets.cfg	file containing all the known targets of the toolset (used by **st200gdb** and **st200run**)
<*tools-dir*>/target/board/<*board_target*>/<*core_target*>/<*endianess*>/<*run-time*> (See *Section 2.2.19 on page 22*).	**bootboard.o** **libboard.a**	the boot code and board specific code for target ***board_target***
<*tools-dir*>/man	<man pages>	

1.4.1 Example command-lines

The ST200 Micro Toolset produces an ST200 object file in ST200 object file formats (elf).

Assuming that we want to compile two files **file1.c** and **file2.c** into an ST220, little endian executable **a.elf**, the set of commands to issue is:

```
$[1] st200cc -c file1.c
$[2] st200cc -c file2.c
$[3] st200cc -o a.elf file1.o file2.o
```

This assumes that the **$PATH** environment variable has been updated with **<*tools-dir*>/bin**.

Command [1] causes the following steps to be executed:

```
<tools-dir>/bin/st200cc                                    # st200cc driver
<tools-dir>/lib/cpp <cpp_flags> file1.c file1.i            # C preprocessor
<tools-dir>/lib/cmplrs/<C compiler> <C Compiler flags> file1.i file1.s
                                                           # C compiler
<tools-dir>/bin/st200as <st200as_flags> file1.s file1.o # ST200
                                                             # Assembler
```

Command [2] causes the following steps to be executed:

```
<tools-dir>/bin/st200cc                                    # st200cc driver
<tools-dir>/lib/cpp <cpp_flags> file2.c file2.i            # C preprocessor
<tools-dir>/lib/cmplrs/<C compiler> <C Compiler flags> file2.i file2.s
                                                            # C compiler
<tools-dir/bin/st200as <st200as_flags> file2.s file2.o # ST200
                                                               # Assembler
```

Command [3] causes the following steps to be executed:

```
<tools-dir>/bin/st200cc                          # st200cc driver
<tools-dir>/bin/st200ld <st200ld_flags>          # ST200 linker
<tools-dir>/lib/st220/le/bare/crt1.o             # ST200 startup files
<tools-dir>/lib/st220/le/bare/crti.o             # ST200 startup files
<tools-dir>/lib/st220/le/bare/crtbegin.o            # ST200 startup files
file1.o file2.o                                  # Application objects

<tools-dir>/lib/st220/le/bare/libc.a              # ST200 C libraries
                                                  # run-time
<tools-dir>/lib/st220/le/bare/libgcc.a               # arithmetic library
<tools-dir>/lib/st220/le/bare/crtn.o             # ST200 startup files
<tools-dir>/lib/st220/le/bare/crtend.o           # ST200 startup files
-o a.elf                                            # Output binary
```

Once steps [1] to [3] are completed, an ST200 executable binary **a.elf** is generated. This can be executed in two different ways:

- Use the stand-alone driver for the load/run tool (available as **st200run**) in the following way:

  ```
  $[4] st200run --target=sim220 a.elf
  ```

  This causes the **a.elf** ST200 binary to be “interpreted” by the **st200run** command. The simulator also provides some minimal cycle counting and statistics facilities.

- Use the ST200 **gdb** debugger:

  ```
  $[4] st200gdb -nw a.elf
  ```

  This causes the ST200 version of **gdb** running on the host platform to load the **a.elf** image.

  Once started, the minimal sequence of commands to issue is:

  ```
  (gdb)> target gdi -t sim220        #inform gdb to use the
                                     #built-in simulator
  (gdb)> load                        #load the binary to the
                                     #simulator environment
  (gdb)> run                         #start the execution
  ```

  The version of **st200gdb** shipped with the ST200 emulation toolchain is a fully functional debugger. If the binary is compiled with the **-g** flag, source-level debugging is possible. Otherwise, assembly level debugging is always supported for ST200 binaries. The **st200gdb** debugger has both a graphical user interface as well as a command-line interface (supported from a terminal shell). The facilities provided by the **st200gdb** debugger are introduced in the *Getting Started* chapter of the *ST200 Cross Development Manual*.

2 st200cc

2.1 Getting started

The **st200cc** compiler is similar to any command-line compiler. It is either invoked from a command line interpreter or from a Makefile and implicitly recognizes files by their extension.

2.1.1 Invoking the compiler

The C compiler is invoked using the **st200cc** command:

```
st200cc {<argument>}
```

where:

```
<argument> = <option> | <input_file>
```

Examples:

```
st200cc -S file.c # produces file.s
st200cc -c file.c # produces file.o
```

Conflicting options are resolved by using the last option on the command line. Any string that is not recognized as an option or a recognized input file is passed down to the link phase.

2.1.2 Input and output

File extension naming conventions are summarized in *Table 1* and *Table 2*.

Extension	Convention
`.c`	C language source file to be pre-processed and compiled
`.h`	C language header file
`.i`	C language source file already pre-processed
`.s`	Assembly language source file to be assembled
`.S`	Assembly language source file to be pre-processed and assembled

Table 1: Input Names conventions

Extension	Convention	Produced by option(s)
`.s`	Assembly language output file	`-S`
`.o`	Object file	`-c`

Table 2: Output Names conventions

The final executable file does not need to have a specific file extension. If no output file name is specified through the **`-o`** option, the executable generated is named **`a.out`**.

Examples:

```
st200cc file.c   # generates the executable a.out
st200cc file.c -o file.u  # generates the executable file.u
```

2.2 Command-line options

2.2.1 Getting help

If the compiler driver is given the **-help** option, it displays the list of available options, and then terminates.

Additionally, the **-help** option can be followed by an additional keyword separated from the help option by a colon. All entries matching the keyword are displayed on the standard output, for example:

```
st200cc -help:-W
```

This command displays all options containing the **-W *string***. In this example, all options related to the emission of compiler warnings will be listed.

2.2.2 Overall options

The options in *Table 3* control the type of processing performed by **st200cc** and the output it generates, for example: an executable, an object file, an assembler file, a pre-processed file, an archive or a dependency list.

Output files produced by these option default to: ***<original_file_name>.<output_extension>*** and can be renamed using the **-o** option.

Option	Description
-c	Compile or assemble the source file, but do not link.
-S	Stop after the compilation phase.
-E	Stop after the preprocessing phase. Output is send to **stdout**.
-v	Print on **stderr** the commands executed to run the compilation phases. The message generated indicates the release identity.
--version	Display the version numbers of the invoked compiler and stop.
-dumpversion	Print the compiler front-end version (for example, 2.95.2) and stop.
-keep	Keep intermediary files produced by the compilation phases.

Table 3: Overall options

2.2.3 C preprocessor options

The preprocessor is run on each C source file before actual compilation. The options in *Table 4* control how the sources are pre-processed.

Option	Description
-E	Only the preprocessor is run.
-C	The preprocessor does not discard comments. Use with the **-E** option.
-P	The preprocessor does discard **#line** information. Use with the **-E** option.
-D*def*	Define the macro definition with the string 1 as the definition.
-D*def*=*defn*	Define the macro definition as ***defn***.
-M	Generates a list of object file dependencies suitable for a Makefile.
-MM	Similar to **-M**, but ignores system header files, that is, header files included by <***header.h***>.
-MG	Along with **-M** or **-MM**, treat missing files as generated in the local directory.
-H	Display the name and path of the header in use.
-dM	Print a list of macros definitions in use after preprocessing. Use with the **-E** option.
-dD	Print a list of macro definitions in use while preprocessing. Use with the **-E** option.
-dN	Same as **-dD**, except that the macro arguments are not shown. Use with the **-E** option.
-fpreprocessed	Indicate to the preprocessor that the input file has already been preprocessed.

Table 4: Preprocessor options

2.2.4 C dialect options

The option **-std=*value*** instructs the compiler front-end to select the appropriate C language dialect to use. For instance, the C99 **restrict** keyword is only recognized with the **-std=c99** option. However, this keyword also exists as a GNU extension keyword, either **__restrict** or **__restrict__** that are recognized by default. Possible values for **-std** are listed in *Table 5*.

Option	Description
-std=iso9899:1990	Same as **-ansi**
-std=iso9899:199409	ISO C as modified in amendment 1
-std=iso9899:1999	ISO C 99
-std=c89	Same as **-std=iso9899:1990**
-std=c99	Same as **-std=iso9899:1999**
-std=gnu89	This is the default, iso9899:1990 + gnu extensions
-std=gnu99	iso9899:1999 + gnu extensions

Table 5: C Dialect options

2.2.5 Warning options

Diagnostic messages can be requested from the compiler to notify potentially erroneous or dangerous C program constructions. *Table 6* lists a subset of the GCC options.

Option	Description
-Wall	Enables all warnings.
-w	Disables all warnings.
-Werror	Turns warnings into errors.
-pedantic	Issues all warnings needed by strict ANSI C compliance.
-pedantic-error	Turn all pedantic warnings into errors.

Table 6: General warning options

The options in *Table 7* take a positive form; a negative form can be systematically constructed by changing the **-W** prefix to a **-Wno** prefix, for example **-Wnoformat**.

Option	Description
-Wchar-subscripts	Warn if an array subscript has type **char**.
-Wformat	Check calls to the **printf** and **scanf** family of library functions.
-Wimplicit-int	Check that all declarations specify a type, which is **int** by default in c89.
-Wimplicit-function-declaration	Warn when a function is used but not declared.
-Werror-implicit-function-declaration	Output error when a function is used but not declared.
-Wimplicit	**-Wimplicit-int** and **-Wimplicit-function-declaration**.
-Wmissing-braces	Warn if an aggregate or union initializer is not fully bracketed.
-Wparentheses	Warn if parentheses are omitted in certain contexts.
-Wreturn-type	Warn when a function is defined with a return-type that defaults to **int**.
-Wswitch	Warn whenever a switch statement may be incomplete.
-Wtrigraph	Warn if any trigraphs are encountered that might change the meaning of the program.
-Wunused	Warn whenever a static function, a label, a parameter, a value is not used.
-Wunknown-pragmas	Warn when a **#pragma** is encountered which is not understood by **st200cc**.
-Wmultichar	Warn if a multi-character constant is used.
-Wshadow	Warn whenever a local variable shadows another variable.
-Wpointer-arith	Warn about anything that depends on the “size of” a function type or of void.
-Wbad-function-cast	Warn whenever a function call is cast to a non-matching type.

Table 7: Detailed warning options

Option	Description
-Wcast-qual	Warn whenever a pointer is cast so as to remove a type qualifier from the target type.
-Wcast-align	Warn whenever a pointer is cast such that the required alignment of the target is increased.
-Wwrite-strings	Warn when trying to write to a string constant.
-Wconversion	Warn if a prototype causes a type conversion that is different from what would happen in the absence of a prototype.
-Wsign-compare	Warn when a comparison between signed and unsigned values could produce an incorrect result.
-Waggregate-return	Warn if any functions that return structures or unions are defined or called.
-Wstrict-prototypes	Warn if a function is declared or defined without specifying the argument types.
-Wmissing-prototypes	Warn if a global function is defined without a previous prototype declaration
-Wmissing-declarations	Warn if a global function is defined without a previous declaration.
-Wmissing-noreturn	Warn about functions which might be candidates for attribute **noreturn**.
-Wredundant-decls	Warn if anything is declared more than once in the same scope.
-Wnested-externs	Warn if an **extern** declaration is encountered within a function.
-Wlong-long	Warn if **long long** type is used. Only active along with **-pedantic**.
-Wpacked	Warn if a structure is given the packed attribute, but the packed attribute has no effect on the layout or size of the structure.
-Wpadded	Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure.

Table 7: Detailed warning options

2.2.6 Debugging options

The **-g** option instructs **st200cc** to generate symbolic information for debugging. DWARF2 format is used.

Note: The **-g** *option may be used with optimization up to level* **-O2** *and with* **-Os** *(see Section 2.2.9: Optimization options on page 16).*

2.2.7 Profiling options

The **-pg** option instructs **st200cc** to generate profiling information, see *Section 5.6: Profiling on page 80*.

Note: Using the **-pg** *option together with* **-g** *enhances the precision of the profile symbolic information.*

2.2.8 Call Trace instrumentation options

The **-finstrument-functions** and **-minstrument-calls** instructs **st200cc** to generate instrumentation calls, see *Section 5.7: Call trace instrumentation on page 81*.

2.2.9 Optimization options

The options in *Table 8* control optimization levels.

Option	Description
-O0	No optimization
-O1	Minimal optimization
-Os	Optimization without code size expansion
-O2	Global optimization
-O3	Aggressive optimization

Table 8: Optimize options

Note: **-O** *optimization is equivalent to* **-O2**. **-Os** *optimization applies the optimizations of* **-O2**, *except for those that increase the code size (such as unrolling).*

The options in *Table 9* enable finer control of the optimization level.

Option	Description
-f[no-]strict-aliasing	**-fstrict-aliasing** allows the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C and C++ this activates optimizations based on the type of expressions. In particular an object of one type is assumed never to reside at the same address as an object of a different type, unless types are almost the same (the aliasing rules are stated in the ANSI C standard, in clause 6.5 (7) Expressions. For example an **unsigned int** can alias an **int**, but not a **void *** or a **double**. A character type may alias any other type. The default is **-fno-strict-aliasing**, for legacy reasons. In future releases, the **-fstrict-aliasing** option will be used by default.
-f[no-]unroll-loops	**-funroll-loops** forces loop unrolling. This is the default at **-O2** and **-O3**. Loops are unrolled up to 4 times at **-O2**, and up to 8 times at **-O3**, with a limit on code size expansion. Loops with a **#pragma unroll** directive are not affected by this option. **-fno-unroll-loops** disable loop unrolling. This is the default at **-Os**. Loops with a **#pragma unroll** directive are not affected by this option.

Table 9: Advanced Optimization options

2.2.10 Code generation options

The options in *Table 10* control various aspects of the code generation.

Option	Description
-fb <*name*>	The feedback file to use for feedback instrumentation. See *Section 5.8: Profiling feedback optimization (PFO) on page 83*.
-fb_create <*name*>	Generate an executable with feedback instrumentation. See *Section 5.8: Profiling feedback optimization (PFO) on page 83*.

Table 10: Code generation options

Option	Description
-fsigned-char **-funsigned-char**	**-fsigned-char** implements type '**char**' as signed. **-funsigned-char** implements **char** as unsigned. Note that when the **-funsigned-char** option is used, the **__CHAR_UNSIGNED__** preprocessor symbol is defined. The compiler default is signed.
-fsigned-bitfields **-funsigned-bitfields** **-fno-signed-bitfields** **-fno-unsigned-bitfields**	These options control whether a bitfield is signed or unsigned, when the declaration does not use either 'signed' or 'unsigned'. The compiler default is signed.
-ffixed-reg=<*register-list*>	**<*register-list*>** is a list of one or several comma-separated register names or dash-separated register ranges, either general purpose registers or boolean registers. The syntax used for a general register is **r*i*** where ***i*** can be 0 to 63 and **b*j*** where ***j*** can be from 0 to 7. This option makes the given registers fixed registers; that is the code generated by the compiler never uses them. There are however, some registers that are used by the compiler for ABI register conventions. See the table of general registers in the *ST200 Run-time Architecture Manual*. Those registers with a specified use must not be reserved with this option. Note that specific care must be taken when using this option since low-level library and run-time support code are not specifically built to support non-ABI register usage. For instance, reserving the **r62** register does not prevent the already compiled library code from using it. Using this option generally requires rebuilding a set of libraries either with the same option (for C/C++ code) or to take into account that this option has been used. Examples: **st200cc -ffixed-reg=r62,b0** **st200cc -ffixed-reg=r24-r62**

Table 10: Code generation options

Option	Description
-f[no-]zero-initialized-in-bss	**st200cc** by default puts variables that are initialized to zero into the **.bss** section that is cleared upon startup. Note that the **-fno-zero-initialized-in-bss** option turns off this behavior because some programs explicitly rely on variables going to the data section. If **-fno-zero-initialized-in-bss** is specified then zero-initialized variables are placed in the data section. The default is **-fzero-initialized-in-bss**.
-fdismissible-load **-fno-dismissible-load**	The **-fdismissible-load** option globally enables and the **-fno-dismissible-load** globally disables the dismissible load generation in all compiler phases. Dismissible loads are only generated at **-Os**, **-O2** and **-O3** optimization levels. Libraries, for example **libc**, **libgcc** are generally built with **-O2** or **-Os** options and thus may include legitimate dismissible loads. This is the case even if **-fno-dismissible-load** is used to compile the application. The **-fno-dismissible-load** option reduces performance and does not reduce code size. This option is intended to be used by OS or library writers and should **not** be used by the general user. Dismissible load generation is enabled by default.
-fverbose-asm **-fno-verbose-asm**	The **-fno-verbose-asm** removes extra commentary information in the generated assembly code. The default is to have verbose **asm** output.

Table 10: Code generation options

2.2.11 -OPT options

The options **-OPT:unroll_size**, **-OPT:cray_ivdep** and **-OPT:liberal_ivdep** modify the behavior of pragmas and are documented in *Section 4.2.1: #pragma unroll on page 49* and *Section 4.2.2: #pragma ivdep on page 50*.

The **-OPT:alias** option is documented in *Section 5.3: Memory dependences in C programs on page 69*.

2.2.12 -Data prefetching options

The option **-mauto-prefetch** triggers automatic data prefetching which is documented in *Section 5.5: Data prefetching on page 75*.

2.2.13 Inlining options

The **-inline**, **-noinline** and **-INLINE** options are provided to control inlining of functions. They are listed in *Table 26 on page 61* and *Table 27 on page 62* and described in *Section 5.2.1: Single file inlining on page 59*.

Only functions marked with the inline keyword are subject to inlining unless specified otherwise.

2.2.14 Interprocedural analysis

The **-ipa** option enables interprocedural analysis, and is described in *Section 5.9 on page 86*. This section documents a range of advanced **-IPA** options that provide control over the optimizations performed.

2.2.15 Symbol visibility specification

The options **-fvisibility** and -**mvisibility-decl** are provided to specify symbol visibility, they are documented in *Section 5.10: Symbol visibility specification on page 89*.

2.2.16 Sending options to a specific phase

The **-W<*phase*>,<*arg*>** option passes the specified argument **<*arg*>** to a specific processing phase **<*phase*>** of **st200cc**.

Table 11 lists the different values of **<*phase*>**.

Value of *phase*	Description
p	Preprocessor **cpp**
f	Compiler front-end
a	Assembler **st200as**
l	Linker **st200ld**
o	Binary optimizer tool **binopt**

Table 11: Possible value for *phase*

There **must** be a comma between the option **-W<*phase*>** and the argument and no spaces. Anything occurring after a space is treated as the next option to **st200cc**. Also the argument is only passed to **<*phase*>** if **<*phase*>** is normally run from the specified command. For example:

```
st200cc -O3 -Wo,--help a.out
```

This command executes the ICache optimizer, which is enabled by the **-O3** option and the argument **--help** is passed in.

Multiple arguments must be separated by commas, with no spaces:

```
st200cc -O3, -Wo,--dump,--static a.out
```

Full details of **binopt** options are given in *Chapter 10: ICache / dead code optimization*.

2.2.17 Specifying the run-time environment

The **-mruntime=*name*** option selects the run-time environment for the compilation and link. ***name*** may be either **bare** to select a bare machine or **os21** to select the OS21 run-time libraries. The default is a bare machine.

To select a run-time other than the default, the **-mruntime** option **must** be used for both the compilation and link.

The *OS21 for ST200 User Manual* describes the use of OS21 for ST200 cores.

2.2.18 Selecting the endianness

The **-EL** option selects code generation for a little endian core, and propagates this option to all the tools that are invoked by the compiler. The startup and libraries are also selected based on this information to be little endian. This is the default.

The **-EB** option selects the big endian mode for compiler code generation, startup and libraries selection.

2.2.19 Configuration options

Options are provided which enable system configuration parameters (called sysconf parameters) or a boot code file to be selected at link time. These options and the modules they select to be linked are described in more detail in the *Getting started* chapter of the *ST200 Cross Development Manual*; they are listed in *Table 12*.

Option	Description
-mboard=<*board_target*>	Select a board target. **bootboard.o**, **libboard.a** and **board.x** located in **target/board/<*board_target*>/<*core_target*>/<*endianess*>/<*runtime*>** are linked.
-mcore=<*core_target*>	Selects the ***core_target*** code generation mode in the compiler. Available targets are **st220** and **st231**. **bootcore.o**, **libcore.a** and **core.x** located in **target/core/<*core_target*>/<*endianess*>/<*runtime*>** are linked.
-msoc=<*soc_target*>	Select a SOC target. **bootsoc.o**, **libsoc.a** and **soc.x** located in **target/soc/<*soc_target*>/<*core_target*>/<*endianess*>/<*runtime*>** are linked.

Table 12: Configuration options

2.2.20 Directory and library options

Table 13 lists the options that select header files, libraries and compiler executables.

Option	Description
-Idirectory	Add directory to the beginning of the search list for include files.
-nostdinc	No predefined include search path.
-l<*library*>	Search the library named **lib<*library*>.a** when linking. The linker looks for the library in the directories specified by the **-L** options and then in a standard list of directories. The position of this option on the command line makes a difference. The linker processes object files and libraries in the order that they are specified on the command line. For example, if the following is specified: **st200cc file1.o file2.o -lmylib** then the files will be processed in the order **file1.o**, **file2.o**, **libmylib.a**. However, if the following is specified: **st200cc file1.o -lmylib file2.o** then the files will be processed in the order **file1.o**, **libmylib.a**, **file2.o**. In this case, **file2.o** should not refer to any symbols defined in **libmylib.a**.
-Ldirectory	Add directory to the beginning of the search list for library files.
-nostdlib	No predefined libraries search path.

Table 13: Directory options

The search path for the various phases of the compiler can be overridden by using the option: **-Y<*phase*>,<*path*>**

where <***phase***> can take the values listed in *Table 11* and <***path***> is the path of the required tool. There **must** be a comma and no spaces separating **-Y<*phase*>** and **<*path*>**.

2.2.21 Environment variables

Currently there are no special environment variables that affect **st200cc**.

2.3 Predefined macros

Predefined macros are described in *Table 14*.

Note: 1 The list of macros currently defined can be obtained by typing:

st200cc -E -dM *filename.c*

*where **filename.c** can be any **.c** file including an empty file.*

2 Do not rely on a macro that is not documented, even if it is currently defined.

3 Some macros values are subject to change along with compiler evolutions, for instance front-end identifications values.

Name	Default definition	Purpose	See also
__open64__	Defined	Compiler technology identification	
__GNUC__	2	Front end major release identification	**-no-gcc**
__GNUC_MINOR__	95	Front end minor release identification	**-no-gcc**
__ST200CC__	Defined, value depends on major compiler version	Compiler identification	
__ST200CC_MINOR__	Defined, value depends on minor compiler version	Compiler identification	
__ST200CC_PATCHLEVEL__	Defined, value depends on compiler patch level	Compiler identification	

Table 14: Predefined macros

Name	Default definition	Purpose	See also
__ST200CC_DATE__	Defined, value depends on compiler release date	Compiler identification	
__ST200CC_VERSION__	Defined, value is an identification string	Compiler identification	
__ST200	Defined	Architecture identification	
__ST200__ **__st200__**	Defined	Architecture identification	
__ST220__ **__st220__**	Defined by default and when **-mcore=st220**	Architecture identification	**-mcore**
__ST231__ **__st231__**	Defined when **-mcore=st231**	Architecture identification	**-mcore**
__LITTLE_ENDIAN__	Defined by default and when using the **-EL** option	Endianness identification	**-EL**
__BIG_ENDIAN__	Defined by the **-EB** option	Endianness identification	**-EB**
_LANGUAGE_C	Defined for C source	Language currently processed is C language.	
_LANGUAGE_ASSEMBLY	Defined for ASM source	Language currently processed is assembly language.	
__STRICT_ANSI__	Defined when **-std=c89** or **-std=c99** or **-ansi**	Compiler is in strict ansi mode.	**-std**

Table 14: Predefined macros

Name	Default definition	Purpose	See also
__STDC_VERSION__	Defined when **-std=c99** with value **199901L**	Compiler is in C99 ansi mode	**-std**
__OPTIMIZE__	Defined as soon as optimization is on.	Optimization mode detection.	**-O**
__INLINE_INTRINSICS	Defined	Intrinsics inlining mode detection.	**-OPT:inline_ intrinsics**
__STDC_HOSTED__	Defined by default.	Hosting mode.	**-f[no-]hosted -f[no-]freestanding**
__BARE_BOARD__	Defined by default	Run-time support identification	**-mruntime**
__OS21_BOARD__	Defined when **-mruntime=os21**	Run-time support identification	**-mruntime=os21**

Table 14: Predefined macros

Note: That the C standard guarantees that the **__cplusplus** *symbol is never defined when compiling C source code.*

2.4 C99 support

The **st200cc** compiler supports a subset of the C99 standard. Most features are implicitly available through default compiler command line options, with the notable exception of the **restrict** keyword that requires the **-std=c99** command line option to be specified.

It is recommended that any code fragment that depends upon C99 specific behavior be guarded by the following preprocessing definitions, which are correctly triggered when the **-std=c99** command line option is used:

```
#if defined(__STDC_VERSION__) && (__STDC_VERSION__ >= 199901L)
// Your C99 dependent code here
#else
#error "This source file depends upon C99 features not available
with this compiler."
#endif
```

Table 15 summarizes the status of the **st200cc** compiler C99 support.

Feature as described in the C99 standard	Status
Restricted character set support via digraphs (**<iso646.h>**)	NO: include file not provided.
Wide character library support	NO: type not supported and library not provided.
More precise aliasing rules via effective type	YES: provided that the **-fstrict-aliasing** option is used
Restricted pointer	YES
Variable length arrays	PARTIAL: only local allocation, but no other features
Flexible array members	NO
Static and type qualifiers in parameter array declarators	NO
Complex support (**<complex.h>**)	NO
Type generic math macros (**<tgmath.h>**)	NO: include file not provided

Table 15: C99 support in st200cc

Feature as described in the C99 standard	Status
The **long long int** type and library functions	YES but C libraries may not provide full support
Increased minimum translation unit	YES
Additional floating-point characteristics (**<float.h>**)	NO
Remove implicit **int**	NO, but can get warning
Reliable integer division	YES
Universal character names	NO
Extended identifiers	NO
Hexadecimal floating-point constants	YES, but no **%a %A printf scanf**/**scanf** conversion specifiers
Compound literals	YES
Designated initializers	YES
// comments	YES
Extended integer type and library functions in **<inttypes.h>** and **<stdint.h>**	NO: include file not provided
Remove implicit function declaration	NO, can get warning
Preprocessor arithmetic done in **intmax_t**/**uintmax_t**	YES
Mixed declaration and code	NO
New block scope for selection and iteration statements	NO
Integer constant type rules	NO
Integer promotion rules	NO
vararg macro	YES
The **vscanf** family of function in **<stdio.h>** and **<wchar.h>**	NO: library support not provided
Additional math library functions in **<math.h>**	NO
Floating point environment access in **<fenv.h>**	NO

Table 15: C99 support in st200cc

Feature as described in the C99 standard	Status
ISO 60559 Arithmetic support	NO
Trailing comma allowed in **enum** declaration	YES
%lf conversion allowed in **printf**	NO
Inline functions	YES but not fully ansi compliant in the extern inline case
The **snprintf** family of functions in **<stdio.h>**	NO: library support not provided
Boolean type in **<stdbool.h>**	NO bool native type but **<stdbool.h>** header provided
Idempotent type qualifiers	YES, but still emits warnings
Empty macro arguments	YES
New struct type compatibility rules	NO
Additional predefined macro names	MOST
_Pragma preprocessing operator	YES
Standard pragmas	NO
__func__ predefined identifier	YES
VA_COPY macro	NO
Additional **strftime** conversion specifiers	NO: library support not provided
LIA compatibility annex	NO
Deprecate **ungetc** at the beginning of a binary file	NO
Remove deprecation of aliased array parameters	NO

Table 15: C99 support in st200cc

3 st200c++

3.1 Introduction to st200c++

The toolset features a GNU C++ 3.3.3 front-end with full library support, but no exceptions support.

It is therefore recommended that you always compile with the **`-fno-exceptions`** option.

The current known limitations are documented, they will be removed in future releases.

This C++ compiler aims at improving performance and compliance with industrial standards (ISO/IEC 14882, Common Vendor ABI, SYSV ELF).

*Note: Using the **st200c++** compiler is very similar to using **st200cc**, therefore only the new options, definitions and usage rules are documented in this chapter.*

3.1.1 References

The ISO/IEC 14882:1998 C++ standard is available from your local ISO or ANSI outlet.

A useful source of information concerning the C++ standard is http://www.open-std.org/JTC1/SC22/WG21.

The GNU Compiler documentation can be found at http://gcc.gnu.org.

The Common vendor ABI is described at http://www.codesourcery.com/cxx-abi.

3.2 Getting started

The **st200c++** compiler is similar to any command-line compiler. It is either invoked from a command line interpreter or from a Makefile and implicitly recognizes files by their extension. It recognizes all **st200cc** compiler options, therefore this chapter only lists the specific C++ additions.

3.2.1 Invoking the compiler

The C++ compiler is invoked using the **st200c++** command:

```
st200c++ {<argument>}
```

where:

```
<argument> = <option> | <input_file>
```

Examples:

```
st200c++ -S -fno-exceptions file.cxx # produces file.s
st200c++ -c -fno-exceptions file.cxx # produces file.o
```

Conflicting options are resolved by using the last option on the command line. Any string that is not recognized as an option or a recognized input file is passed down to the link phase.

3.2.2 Input and output

File extension naming conventions are summarized in *Table 16*.

Extension	Convention
`.cxx`, `.cpp`, `.c++`, `.C`, `.CC`, `.CPP`, `.CXX`	C++ language source file to be pre-processed and compiled.
`.ii`	C++ language source file already pre-processed.

Table 16: Input names conventions

The final executable file does not need to have a specific file extension. If no output file name is specified through the `-o` option, the executable generated is `a.out`. Examples:

```
st200c++ file.cxx    # generates the executable a.out
st200c++ file.cxx -o file.u  # generates the executable file.u
```

3.3 C++ specific command-line options

3.3.1 C++ dialect options

The option **-std=*value*** instructs the compiler front-end to select the appropriate C++ language dialect to use. Possible value for **-std** are listed in *Table 17*.

Option	Description
-std=c++98	Default if **-fansi**. Set language to ISO/IEC 14882.
-std=gnu++98	Default. Set language to ISO/IEC 14882 and support GNU extensions.

Table 17: C++ dialect options

3.3.2 C++ specific warning options

Diagnostic messages can be requested from the compiler to notify potentially erroneous or dangerous C++ program constructions. *Table 18* lists a subset of the GCC options.

Option	Description
-W[no-]abi	Warn when compiler generates code that is probably not compatible with the vendor-neutral C++ ABI.The default is **-Wabi**.
-W[no-]ctor-dtor-privacy	Warn when a class seems unusable because all the constructors or destructors in that class are private. The default is **-Wctor-dtor-privacy**
-W[no-]effc++	Warn about violations of style guidelines from Scott Meyer's 'Effective C++ book'. The default is **-Wno-effc++**.
-W[no-]old-style-cast	Warn if an old-style cast to a non-void type is used within a C++ program. The default is **-Wno-old-style-cast.**

Table 18: C++ warning options

Option	Description
-W[no-]overloaded-virtual	Warn when a function declaration hides virtual functions from a base class. The default is **-Wno-overloaded-virtual.**
-W[no-]deprecated	Do not warn about usage of deprecated features. The default is **-Wdeprecated.**
-W[no-]pmf-conversions	Disable the diagnostic for converting a bound pointer to a member function to a plain pointer. The default is **-Wpmf-conversions.**
-W[no-]reorder	Warn when the order of member initializers given in the code does not match the order in which they must be executed. The default is **-Wno-reorder**

Table 18: C++ warning options

The options in *Table 18* take a positive form; a negative form can be systematically constructed by changing the **-W** prefix to a **-Wno** prefix, for example **-Wnoformat**.

3.3.3 Code generation options

The options in *Table 19* control various aspects of the code generation.

Option	Description
-f[no-]rtti	Enable (or disable) generation of rtti information.
-f[no-]exceptions	Enable (or disable) exception handling.
-f[no-]check-new	Check that the pointer returned by operator new is non null before attempting to modify the storage allocated or calling the object's constructor.
-f[no-]implicit-templates	Emit code (or never emit code) for non-inline templates which are instantiated implicitly.
-f[no-]access-control	Turn on (or off) all access checking.

Table 19: C++ code generation options

Option	Description
-f[no-]elide-constructors	Omit creating a temporary which is only used to initialize another object of the same type. **-fno-elide-constructors** forces the compiler to call the copy constructor even when it could be optimized out.
-f[no]-for-scope	Under **-fno-for-scope**, the scope of variables declared in a **for-init-statement** extends to the end of the enclosing scope. Under **-ffor-scope**, the scope of variables declared in a for-init-statement is limited to the **for** loop itself, as specified by the C++ standard. The default is **-ffor-scope.**
-f[no]-gnu-keywords	When using **-fno-gnu-keywords**, **typeof** is not recognized as a keyword, so that code can still use this word as an identifier. The keyword **__typeof__** can still be used instead. The default is **-fgnu-keywords.-ansi** implies **-fno-gnu-keywords**.
-f[no]-permissive	**-fpermissive** downgrades some diagnostics about non-conformant code, allowing some non-conformant code to compile. The default is **-fno-permissive**.
-ftemplate-depth=*n*	Set the maximum instantiation depth for template classes to ***n***.
-fmessage-length=*n*	Try to format error messages so that they fit on lines of about ***n*** characters.
-fdiagnostics-show-location=[once|every-line]	Instructs the diagnostic messages reporter to emit ‘once’ or ‘every-line’ source location information.

Table 19: C++ code generation options

3.3.4 Directory and library options

Table 20 lists the options that select header files, libraries and compiler executables.

Option	Description
-nostdinc ++	No predefined include search path.

Table 20: Directory options

3.4 Predefined macros

Predefined macros are described in *Table 21*.

Note: 1 The macros defined by the C compiler are still relevant unless otherwise stated (for example the macros defining the endianness and the core)

2 The list of macros currently defined can be obtained by typing:

st200c++ -E -dM *filename.cxx*

*where **filename.cxx** can be any **.cxx** file including an empty file.*

3 Do not rely on a macro that is not documented, even if it is currently defined.

4 Some macros values are subject to change in future compiler versions, for instance front-end identification values.

Name	Default definition	Purpose	See also
__GNUC__	3	Front end major release identification	**-no-gcc**
__GNUC_MINOR__	3	Front end minor release identification	**-no-gcc**
__GNUC_PATCHLEVEL__	3	Front end minor release identification	**-no-gcc**

Table 21: Predefined macros

Name	Default definition	Purpose	See also
__GNUG__	3	C++ Front end major release identification	**-no-gcc**
_LANGUAGE_C_PLUS_PLUS	Defined only for C++ source	Language currently processed is C++ language.	
__cplusplus	Defined for C++ source	Language currently processed is C++ language.	
__STRICT_ANSI__	Defined when **-std=c++98 -ansi**	Compiler is in strict ansi mode.	**-std**
__EXCEPTIONS	Defined, unless **-fno-exceptions** is used	Compiler supports exceptions	**-f[no-]exceptions**
__DEPRECATED	defined, unless **-Wno-deprecated** is used	Compiler supports legacy code	**-Wno-deprecated**
__GXX_ABI_VERSION	102	C++ abi version	

Table 21: Predefined macros

Note: The C standard guarantees that the **__cplusplus** *symbol is never defined when compiling C source code, and the C++ standard guarantees that* **__cplusplus** *is defined when compiling C++ source code.*

3.5 GNU C++ language extensions

3.5.1 Extensions to the C++ language

3.5.1.1 C language extensions supported in C++

The following C language extensions are supported in C++:

- **alignof**,
- **typeof**,
- case ranges,
- label as values.

3.5.1.2 C++ specific extensions

Minimum and maximum operators

a <? b is the minimum, returning the smaller of the numeric values a and b.

a >? b is the maximum, returning the larger of the numeric values a and b.

3.5.2 Variable, functions and type attributes

3.5.2.1 C language attributes supported in C++

The following C++ attributes are supported in the C++ context.

Placement and layout attributes

- **section**,
- **weak**,
- **alias**,
- **align**,
- **pack**,
- **constructor**,
- **destructor**.

Optimization attributes

- **malloc**,
- **noreturn**.

3.5.2.2 C++ specific attributes

Initialization priority

Note: This feature is unsupported in the current release.

Support '**__attribute__ ((init_priority (*n*)))**' for controlling the order of initialization of file-scope objects.

Where ***n*** is a number representing the priority of that object. File-scope objects are initialized in order from the lowest number (highest priority) to the highest (lowest priority).

3.5.3 Built-ins

The following C language built-ins are supported in C++:

- **__builtin_constant_p**,
- **__builtin_expect**.

3.6 GNU C++ asm statement

The GNU asm statement is supported as in the C compiler.

3.7 Standard Template Libraries (STL) support

This support is based on libstdc++-v3, you can consult the documentation online at the following site:

http://gcc.gnu.org/onlinedocs/libstdc++/documentation.html

The very latest user documentation is available through:

http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-3.4/index.html

The full C++ standard library is supported, excepted for the **<complex>** header and limited to the underlying C library support limitations (no **wide char** support), but augmented with the **long long** support.

The library includes the GNU STL extensions and the backward compatibility headers.

3.7.1 Standard headers status

All the following standard headers are supported, unless otherwise stated:

Header files	
<algorithm>	**<iomanip>**
<bits>	**<ios>**
<bitset>	**<iosfwd>**
<cassert>	**<iostream>**
<cctype>	**<istream>**
<cerrno>	**<iterator>**
<cfloat>	**<limits>**
<ciso646>	**<list>**
<climits>	**<locale>**
<clocale>	**<map>**
<cmath>	**<memory>**
<complex>	**<new>**
<csetjmp>	**<numeric>**
<csignal>	**<ostream>**
<cstdarg>	**<queue>**
<cstddef>	**<set>**
<cstdio>	**<sstream>**

Table 22: STL standard header files

Header files	
<cstdlib>	<stack>
<cstring>	<stdexcept>
<ctime>	<streambuf>
<cwchar>	<string>
<cwctype>	<typeinfo>
<deque>	<utility>
<exception>	<valarray>
<fstream>	<vector>
<functional>	

Table 22: STL standard header files

3.7.2 Supported extension headers:

All these extensions live in the '**__gnu_cxx**' namespace (compared with the '**std**' namespace of the standard library functions).

Note: Using these extensions can be useful, however, it may restrict portability, because not all compilers or libraries support these extensions.

Header files	
<ext/algorithm>	<ext/memory>
<ext/functional>	<ext/numeric>
<ext/hash_map>	<ext/rope>
<ext/hash_set>	<ext/slist>
<ext/iterator>	

Table 23: Supported extension headers

3.7.3 Supported backward compatibility headers:

All the following backward compatibility headers are supported, unless otherwise stated.

Using these files directly is not recommended, the compiler warns about this usage as being deprecated. To remove this warning, use the **-Wno-deprecated** compiler option.

Header files	
<algo.h>	<map.h>
<algobase.h>	<multimap.h>
<alloc.h>	<multiset.h>
<bvector.h>	<new.h>
<complex.h>	<ostream.h>
<defalloc.h>	<pair.h>
<deque.h>	<queue.h>
<fstream.h>	<rope.h>
<function.h>	<set.h>
<hash_map.h>	<slist.h>
<hash_set.h>	<stack.h>
<hashtable.h>	<stream.h>
<heap.h>	<streambuf.h>
<iomanip.h>	<strstream.h>
<iostream.h>	<tempbuf.h>
<istream.h>	<tree.h>
<iterator.h>	<vector.h>
<list.h>	

Table 24: Backward compatibility headers

3.7.4 Identifying the STL version

The version supported is defined numerically through **__GLIBCPP__** and as a string through **_GLIBCPP_VERSION**. With the current release, the value of **__GLIBCPP__** is **20040214** and the value of **_GLIBCPP_VERSION** is **3.3.3**. These versions are consistent with the C++ front-end used.

A simple way to show it is to compile and run the following program:

```
#include <cstdio>
int main(int argc, char *argv[])
{
     ::printf("__GLIBCPP__ is %d __GLIBCPP_VERSION__  %is s\n",
          __GLIBCPP__, _GLIBCPP_VERSION) ;
     return 0 ;
}
```

Note: In future releases, this preprocessor definition might be deprecated and replaced by **__GLIBCXX__** *(and* **_GLIBCXX_VERSION***).*

3.8 RTTI notes

The **rtti** support is enabled by default, to disable it, use the **-fno-rtti** compiler option.

When **rtti** is enabled there is:

- support for the **<typeinfo>** header and the **typeid** operator.
- support for **dynamic_cast**.

If **rtti** is disabled:

- the compiler refuses to compile code containing calls to the **typeid** operator,
- code containing calls to **dynamic_cast** has undefined behavior (and typically aborts at run-time).

The limitation is, that as exceptions are not supported, the exception **std::bad_cast** is never thrown in case **dynamic_cast** fails, this is replaced by a call to the **abort()** library function.

3.9 Exceptions notes

Exceptions are not supported. This implies that **all C++ code** must be compiled using the **-fno-exceptions** option. Be aware that even if your code does not use exceptions, the standard headers do, so you must use **-fno-exceptions**.

*Note: The STL generally relies on exceptions. In the absence of exception implementation, the run-time support simply calls **abort()** when an exception is encountered. For instance, a failure to allocate memory through the '**new**' operator goes through a **__throw_bad_alloc** function that in turns calls '**abort**' when the exception support is disabled. The usual stack-unwinding process is not activated, so the objects that would be destroyed automatically through this process are not.*

It is not possible to write exception specifications in function prototypes when the exception support is disabled, except for the empty specification clause **throw()**.

It is possible to have a more explicit abort diagnostic by establishing a signal handler on the **SIGABRT** signal.

The compiler defines the preprocessor macro **__EXCEPTIONS** only when exceptions are enabled. When writing code to work both with and without exception handling support, use this macro to select the appropriate statements.

3.10 Checking operator new return value

When operator **new** is overloaded with a definition that may return 0, it must be declared **throw()** or the user must use the **-fcheck-new** compiler option, otherwise, unexpected control flow will be generated.

Consider the following example where operator **new** is declared **throw()**:

```
#include <cstdio>
namespace Tspace
{
     class T
     {
          public:
               void *operator new(size_t n)
               { return 0; }
               T() { ++ctor; }
               static int ctor;
     };
     int T::ctor = 0;
```

```
}
void surprise() { ::puts("UNEXPECTED\n") ; }
int main(int argc, char *argv[])
{
     using namespace Tspace ;
     T *p = new T();
     if (p != 0)
          surprise();
     else {
          if(T::ctor != 0)
               ::puts("FAILURE") ;
          else
               ::puts("SUCCESS") ;
     }
}
```

Using this example the following command:

$ **st200c++ example.cxx**

outputs the warning:

```
warning: 'operator new' must not return NULL unless it is declared
-output 'throw()' (or -fcheck-new is in effect) -output
```

Running the executable:

$ **st220run -tsim a.out**

produces the following message:

```
FAILURE
```

Compiling the example with the **-fcheck-new** option and running the resulting executable:

$ **st200c++ -fcheck-new example.cxx**

$ **st220run -tsim a.out**

produces the message:

```
SUCCESS
```

3.11 Known limitations

3.11.1 Status of the known limitations

The known limitations are:

- no exception support,
- performance **`#pragmas`** are not recognized,
- no intrinsics support,

4 Pragmas

4.1 Pragmas short description and syntax

Syntax	Scope	Description	Optimization level[a]
`#pragma unroll (unroll_amount)`	Start of a loop body	Unroll the loop *unroll_amount* times	-O2
`#pragma ivdep`	Start of a loop body	Liberalizes dependence analysis	-O3
`#pragma loopdep PARALLEL|VECTOR|LIBERAL`	Start of a loop body	Liberalizes dependence analysis	-O3
`#pragma loopmod(q,r)`	Start of a loop body	Provides trip count modularity information	-O2
`#pragma looptrip(n)`	Start of a loop body	Provide trip count estimation information	-O2
`#pragma pipeline(p,r)`	Start of a loop body	Controls pipeliner	-O3
`#pragma loopseq READ | WRITE`	Start of a loop body	Ordering of the READ (or WRITE) accesses	-O2
`#pragma frequency_hint NEVER|FREQUENT`	Applies to the function or statement that follows the pragma	Execution frequency hint	Any

Table 25: st200cc pragmas

Syntax	Scope	Description	Optimization level[a]
`#pragma ident "string"`	-	Adds a `.comment` section to an assembly file.	Any
`#pragma weak symbol`	-	Marks a symbol as weak	Any
`#pragma inline_next (function)`	Function call site	Inlining [b]	`-O1`
`#pragma noinline_next (function)`	Function call site		
`#pragma inline_function (function)`	Function		
`#pragma noinline_function (function)`	Function		
`#pragma inline_file (function)`	File		
`#pragma noinline_file (function)`	File		
`#pragma defaultinline (function)`	-		

Table 25: st200cc pragmas

a. This column denotes the lowest optimization level for which the pragma has an effect. For example `-O0` means the pragma is applicable even when optimization is switched off. A list of optimization levels is given in *Section 2.2.9: Optimization options on page 16*.

b. All inlining pragmas are described in *Section 5.2.4: Inlining pragmas on page 63*.

4.2 Loop optimization pragmas

4.2.1 #pragma unroll

This pragma suggests to the compiler the type of loop unrolling that should be done. The pragma is a recommendation to the compiler to add ***n*-1** copies of the loop body to the inner loop. The value of ***n*** must be at least **1**. If it is **1**, then unrolling is not performed.

If the loop that this pragma immediately precedes is an inner loop, then it implies standard inner loop unrolling.

Inner loop unrolling example:

```
for (i=0; i < 10; i++)
#pragma unroll (2)
    for (j=0; j < 10; j++)
        a[i][j] = a[i][j]+b[i][j];
```

becomes:

```
for (i=0; i < 10; i++)
    for (j=0; j < 10; j+=2) {
        a[i][j]   = a[i][j]  +b[i][j];
        a[i][j+1] = a[i][j+1]+b[i][j+1];
    }
```

If the loop that this pragma immediately precedes is an outer loop that contains only an inner loop, then outer loop unrolling, followed by the loop fusion of the resulting inner loops, is attempted. This transformation, known as unroll-and-jam, is especially useful to create parallel execution opportunities when the innermost loop alone does not present such opportunities.

Unroll-and-jam example:

```
// Ensure ad[] and sd[] do not alias.
#pragma unroll(2)
for (i=0; i<16; i++) {
    int sum = 0;
    for (k=M; k<8+M; k++) {
        sum += sd[k]*sd[k-i];
    }
    ad[i] = sum;
}
```

becomes:

```
for (i=0; i<16; i+=2) {
    int sum0 = 0;
    int sum1 = 0;
    for (k=M; k<8+M; k++) {
        sum0 += sd[k]*sd[k-i];
        sum1 += sd[k]*sd[k-i-1];
    }
    ad[i] = sum0;
    ad[i+1] = sum1;
}
```

The following tips provide information on how to control the desired inner loop unrolling with the pragma unroll value.

- A counted loop with a compile-time constant trip count is **always** fully unrolled if a pragma unroll with a value greater or equal to the loop trip count is specified.
- When a counted loop is not fully unrolled, the pragma unroll value is rounded to the greatest power of two lower than the specified unrolling value.
- The maximum size of a loop after unrolling is controlled by the command line option **-OPT:unroll_size=*<n>***.

4.2.2 #pragma ivdep

This pragma instructs the compiler to liberalize dependence analysis between memory accesses. The **#pragma ivdep** applies only to inner loops. If it is used on a loop that has an inner loop, the compiler ignores it. By default, this pragma allows the compiler to assume there are no memory dependences between loop iterations.

The following command line options modify the **ivdep** semantic.

- **-OPT:cray_ivdep=TRUE**
  Only ignore backward memory dependences (Cray semantics).
- **-OPT:liberal_ivdep=TRUE**
  Also ignore all memory dependences in the same loop iteration.

Example:

```
#pragma ivdep
for (i = 0; i < n; i++) {
    a[b[i]] = a[b[i]]+3; // These dependences cannot be computed by
                         // the compiler
}
```

4.2.3 #pragma loopdep

This pragma instructs the compiler to liberalize dependence analysis between memory accesses, based on the specified type of loop dependences. Contrary to the pragma **ivdep** described above, the semantics cannot be modified by command line options.

The **loopdep** pragma takes an argument to tell the compiler which kind of loop dependencies it can ignore, **VECTOR**, **PARALLEL** or **LIBERAL**.

#pragma loopdep VECTOR

#pragma loopdep VECTOR allows the compiler to assume there are no backward memory dependences between loop iterations. This pragma is equivalent to **#pragma ivdep, -OPT:cray_ivdep=TRUE**.

Example:

```
#pragma loopdep VECTOR
for (i = 0; i < n; i++) {
    a[i] = a[i+k]+3;
}
```

In this example, the compiler cannot tell when **a[i+k]** does not depend on **a[i]**, but this is in fact the case if **k** is always **> 0** in the program. The pragma allows the compiler to assume there are no dependences between the read of **a[i+k]** in the current loop iteration, and the write of **a[i]** in the following loop iterations. The compiler could rewrite the loop as:

```
for (i = 0; i < n; i+=2) {
    t0     = a[i+k]+3;
    t1     = a[i+1+k]+3;
    a[i]   = t0;
    a[i+1] = t1;
}
```

#pragma loopdep PARALLEL

#pragma loopdep PARALLEL allows the compiler to assume there are no dependences between any two memory accesses that are in different loop iterations. This pragma is equivalent to:

```
#pragma ivdep, -OPT:cray_ivdep=FALSE -OPT:liberal_ivdep=FALSE
```

Example:

```
#pragma loopdep PARALLEL
for (i = 0; i < n; i++)
    a[b[i]] = a[b[i]] + 3;
```

In this example, the compiler cannot tell that either the load or store of **a[b[i]]** in the current loop iteration does not depend on the load or store of **a[b[i]]** in a following loop iteration. This is in fact the case if **b[i] != b[j]** for all **i != j**. The compiler could rewrite the loop as:

```
for (i = 0; i < n; i+=2) {
    t1        = a[b[i+1]] + 3;
    t0        = a[b[i]] + 3;
    a[b[i+1]] = t1;
    a[b[i]]   = t0;
}
```

#pragma loopdep LIBERAL

#pragma loopdep LIBERAL allows the compiler to assume there are no dependences between any two memory accesses that are either in the same, or different, loop iterations. This pragma is equivalent to:

```
#pragma ivdep, -OPT:liberal_ivdep=TRUE
```

Example:

```
#pragma loopdep liberal
for (i = 0; i < n; i++) {
    a[j] = b[i];
    c[i] = a[i] + 3;
}
```

In this example, the compiler cannot tell that the load of **a[i]** does not depend on the store of **a[j]**. This is in fact the case if **i != j** for all values of **i** and **j** in the loop iterations.

4.2.4 #pragma loopmod

This pragma tells the compiler the number of times a loop is taken in terms of a multiple ***q*** and a residual ***r***.

The syntax of this pragma is:

```
#pragma loopmod(q,r)
```

where ***q*** is strictly a positive integer, ***r*** is a positive integer, ***r*** < ***q***.

For example:

```
#pragma loopmod (4,0)
```

This tells the compiler that the loop is taken 4, 8, 12 .... times.

```
#pragma loopmod (4,1)
```

This tells the compiler that the loop is taken 5, 9, 13 .... times.

When applied to an inner loop, this pragma indicates that the trip count **tc**, that is the number of iterations that are executed by any execution of the loop can be written as:

$tc = p\ q + r$ with $p > 0, r >= 0$

Where p is strictly a positive integer. This information helps the compiler in loop unrolling optimization, and in software pipelining.

When unrolling loops, the compiler creates multiple loop bodies (the unrolling factor specifies the number of loop bodies created). However, the compiler cannot always statically determine the trip count. When it cannot determine the trip count, the compiler must also create residual code in case the unrolling factor is not a divisor of the loop trip count.

However, it is possible for application writers to know the modular properties of some of the loops in their own code. Bringing this accurate information to the compiler, the residual code can be largely removed or better optimized.

Note: Bringing inexact information on the trip count may lead to inexact code. Be careful that the property asserted is valid in all cases.

The following example shows the use of the **#pragma loopmod**.

```
void copychar(unsigned char* __restrict p, unsigned char * q,
unsigned int sz)
{
    int i ;
    assert(sz % 4 == 0) ;
#pragma loopmod(4,0)
    for(i=0; i<sz; i++)
        p[i] = q[i];
}
```

The function **copychar** duplicates a byte stream, whose size must be a multiple of 4. During unrolling, and without the pragma, the compiler would create a residual loop. This is totally removed when the pragma information is asserted. In this example, the pragma does not provide the compiler with any information about the memory alignment of **p** or **q**, which the compiler would need to generate word accesses after unrolling.

4.2.5 #pragma looptrip

This pragma instructs the compiler that the estimate of the number of the iterations of the loop (the loop trip count estimate) is ***n***. This is not an assertion that the loop effectively iterates ***n*** times.

A number of optimizations are affected by the #pragma, when the compiler has not already determined the exact trip count:

- basic block frequency estimation uses this information as an approximation of the loop trip count,
- unrolling and cross-iteration optimizations are reduced if the given loop trip count estimate is low,
- software pipelining is limited if the estimate is low,
- automatic data prefetch generation is limited if the estimate is low.

One scenario of usage is for '**for**' loops with trip counts of unknown values where the user knows that the approximate effective value is low:

```
#pragma looptrip(4)
for (i=0; i<n; i++)
a[i] = b[i] ;
```

This example avoids non-beneficial optimizations. On such loops the compiler trip count estimate without the pragma is 100.

A second scenario is for '**while**' loops where the user knows that the approximate effective trip count is high:

```
#pragma looptrip(100)
while (*p++=*s++)
```

This example gives a better approximation of the weight of the loop. Generally the compiler trip count estimate for a '**while**' loop is very low.

Note: Possible error messages are:

- **Warning : pragma 'LOOPTRIP' : inconsistent with computed value, ignored**
- **Warning : pragma 'LOOPTRIP' : not followed by a loop, ignored**
- **Warning : malformed '#pragma looptrip (n)'**

4.2.6 #pragma pipeline

This pragma is used to override pipelining and renaming defaults on a particular loop. This is a **-O3** optimization and the defaults are: **pipelining=3** and **renaming=2**. With **pipelining=3**, the maximum performance software pipeline is built, with possible code increase. Using **pipelining=1** and **renaming=1** may reduce code size.

The meaning for **pipelining** is as follows:

- **pipelining=0**, no loop iteration overlap,
- **pipelining=1**, maximum overlap over 1 iteration,
- **pipelining=2**, maximum overlap over 3 iterations,
- **pipelining=3**, maximum overlap over 7 iterations.

The meaning for **renaming** is as follows:

- **renaming=0**, no register renaming,
- **renaming=1**, register renaming enabled,
- **renaming=2**, local register modulo renaming.

4.2.7 Code generation pragmas

#pragma loopseq READ

#pragma loopseq WRITE

This pragma instructs the compiler that the memory **READ** accesses (or respectively the memory **WRITE** accesses) as they appear in the loop should be sequenced. This is not an assertion that the accesses must be kept in sequence, for instance, this is not a replacement for volatile accesses where it is mandatory to keep them in order.

The effect of this pragma is that the scheduler will serialize all load prefetch operations (or respectively all stores) in the loop. Thus the memory read (or write) accesses, as written in the C code are kept in order, as long as no aggressive transformation occurs in the loop.

A scenario is when the user wants to keep memory writes in order to take advantage of a combining write buffer:

```
#pragma loopseq WRITE
for(i=0; i<n; i++) {
a[i] = ...;
a[i+1] = ... ;
a[i+2] = ... ;
a[i+4] = ... ;
}
```

The pragma hints that the compiler should keep writes to the array in order. If the loop is unrolled, generating a large number of stores, this improves locality and may take advantage of combining write buffers. By default the compiler does not put restrictions on the ordering of non-overlapping store operations.

A second scenario is when the user has scheduled by hand prefetch and load operations and wants to ensure that the compiler does not reorder them.

```
#pragma loopseq READ
for(i=0; i<n; i+=S) {
  ... = a[i] ;
  __builtin_prefetch(&a[i+S]) ;
}
```

The pragma hints that the compiler should keep the load and prefetch in order. In this example, the prefetch is not placed before it is effectively used in the next iteration by the load.

4.2.8 Heuristic pragmas

#pragma frequency_hint

This pragma allows the user to specify information about the execution frequency for certain regions of code with the following frequency specifications:

NEVER This region of code is never or rarely executed. The compiler might move this region of the code away from the normal path. This movement might either be at the end of the procedure or at some point to an entirely separate section.

FREQUENT This region of code is frequently executed. The compiler might try to put this region in the fall through path.

Example:

```
if (debug) {
    #pragma frequency_hint NEVER
    trace();
}
```

4.3 Miscellaneous pragmas

4.3.1 #pragma ident

Adds a **.comment** section in an assembly file.

4.3.2 #pragma weak

Marks a symbol as **weak**.

Instructs the link editor to not issue a warning if it does not find a defining declaration of the specified weak symbol. In which case the symbol is set to **0**.

Allow the overriding of the current definition by a non-weak definition.

```
#pragma weak opt_handler
extern void opt_handler (void);

int main(int argc, char *argv[])
{
    /* If opt_handler has not been defined, the linker does not
complain and the condition is false.*/
    /* If opt_handler has been defined, the opt_handler is
invoked.*/
    if (opt_handler)
        opt_handler();
}
```

5 Optimization guide

5.1 Introduction

This chapter describes specific compiler options and techniques that can be used to gain maximum performance in your application.

5.2 Inlining

Inline function expansion is performed for function calls that the compiler estimates to be frequently executed. These estimations are based on a set of heuristics. The compiler might decide to replace the instructions of the call with code for the function itself (inline the call). The compiler supports both the single file inlining mode as described in *Section 5.2.1* and cross file inlining through the IPA optimization described in *Section 5.9: Interprocedural analysis optimization (IPA) on page 86*.

5.2.1 Single file inlining

The purpose of this section is to make users aware of the underlying algorithms used to select functions to inline. First, it describes how possible candidates are selected for inlining, and how the selection is finalized, taking size conditions into account. Then, user-level compiler switches are listed, to show how the inlining process can be controlled.

The inlining decisions of the compiler can be observed with the **`-INLINE:list`** option, it is recommended that this option should be used when tuning inlining decisions.

There are two kinds of candidates for inlining: **may-inline** and **must-inline** functions.

May-inline functions are selected by the compiler according to the following conditions:

- function is declared with the **`inline`** C keyword,
- the functions not declared inline are may-inline candidates only if the **`-INLINE:only_inline=off`** option is specified. In this case, the following categories are also may-inline candidates:
  - function is declared with the **`static`** C keyword, whose name is not weak, and whose address is neither passed nor saved,
  - finally, if the generated code does not follow the GP-relative model, the non-weak condition is sufficient. This large-span condition is included to add inlining opportunities in the current programming model of the ST200, where there is currently no support for shared objects. As a result, no function is preemptible, except by using the **weak** attribute.

Must-inline functions are specified by the user, through the command line option:

`-INLINE:must=`*`fn1`*`,`*`fn2`*`,...`

May-inline and **must-inline** functions are then checked against several criteria to decide whether to inline them or not.

Inlining criteria

Each candidate function is checked against inlining-exclusion cases which include:

- requires no-inlining by the user (**`-INLINE:never=`*`fn`*`, -INLINE:none`** command line options),
- recursive function,
- **`vararg`** function,
- exception handler.

After this preliminary test, each candidate function is inlined regardless of cost if it is marked **must-inline**, or if the **`-INLINE:all`** option has been specified by the user.

Otherwise, cost evaluation is used to decide whether to inline or not, and the candidate function is rejected if its estimated cost is above a given threshold set by the compiler. The **`-INLINE:list=on`** option can be used to list what is inlined.

Changing the compiler limits is **not** recommended, since this can lead to longer compilation times or increased memory usage or both, with no noticeable performance benefit.

Finally:

- the function to be inlined must be defined and visible in the same source file as the function using it,
- a static function that is inlined can be in specific circumstances considered “dead”, and removed from the final object file.

5.2.2 st200cc inlining options

Table 26 specifies the options to control the stand-alone inlining.

More than one sub-option can be specified to the **-INLINE:** option either by using colons to separate each sub-option or by specifying multiple options on the command line. Some **-INLINE:** sub-options are specified with a setting that either enables or disables the feature. To disable a feature, specify the sub-option with either **=OFF**, **=FALSE** or **=0** (all these strings are case insensitive, for example **-INLINE:list=OFF**). To enable a feature, either use the option name alone (for example **-INLINE:list**) or any other string can be used on the right of the '**=**' sign (as in **-INLINE:list=all**). It is generally recommended to use **=ON**, **=TRUE**, **=1** for the sake of clarity (for example **-INLINE:list=ON**).

Option	Description
-inline	Enable inlining on inline functions. This is activated by default at optimization levels > 1.
-noinline	Disable inlining.
-INLINE:(on|off)	Enable/disable inlining. Use of other **-INLINE** options implicitly set this to on.
-INLINE:all	Bypass cost evaluation when inlining 'may inline' functions. This option conflicts with none, and takes precedence if both are specified. Default is off.
-INLINE:all_inline	Inline all functions marked by the C language **inline** keyword.

Table 26: Standalone inlining options

Option	Description
-INLINE:only_inline=(on|off)	Default is on. Inline only functions marked by the C language **inline** keyword. The **-INLINE:only_inline=off** option is mandatory to allow inlining of non "inline" functions.
-INLINE:static=(on|off)	Default is off. Allow static functions to be candidates for inlining.
-INLINE:aggressive=(on|off)	Inline even non-leaf, out-of-loop calls. Default is off.
-INLINE:list=(on|off)	List compiler actions. Default is off.
-INLINE:must=name1[,name2...]	Always attempt to inline the names subroutines in addition to the default heuristic.
-INLINE:never=name1[,name2...]	Never attempt to inline the names subroutines.
-INLINE:dfe	Allow dead function elimination. Default is on.
-INLINE:specfile=filename	Specifies a filename containing inlining options. Default is none.

Table 26: Standalone inlining options

In addition to these options, the following option may be of interest when building a large body of inline functions (which is not recommended and may adversely affect performance).

Option	Description
-OPT:Olimit=[0..*n*]	Functions larger than size ***n*** are not optimized. Default is 3000. Specifying 0 removes any limit but may lead to a very long compile time.

Table 27: Option changing inlining behavior

5.2.3 Extern inline functions

If both ‘inline’ and ‘extern’ are specified in a function definition, then the definition is used only for inlining. The function is never compiled on its own, not even if its address is referred to explicitly. The address becomes an external reference, as if the function had only been declared but not defined

This combination of ‘inline’ and ‘extern’ has almost the same effect as a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking ‘inline’ and ‘extern’) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any instances of the function remain, they will refer to the single copy in the library.

5.2.4 Inlining pragmas

5.2.4.1 Introduction

The inlining process can be controlled within the C source code using **`#pragmas`**.

The **st200cc** compiler already supports several command-line options to configure its behavior, but it is not flexible enough. For instance, with the option **`-INLINE:never=foo`** the user can disable the inlining of **`foo`** everywhere it is called; conversely, with **`-INLINE:must=foo`** the user can force inlining of **`foo`** everywhere.

The user has the ability to force inlining or non-inlining at call sites through the use of pragmas. In addition, the **`noinline`** and **`always_inline`** attributes can be used at function declaration.

5.2.4.2 Pragmas

To force inlining or non-inlining of a function in the scope of a call site, the following two pragmas are introduced:

- **`#pragma inline_next (foo,...)`** forces inlining of function **`foo`** in the next statement,
- **`#pragma noinline_next (foo,...)`** prevents inlining of function **`foo`** in the next statement.

The **`...`** denotes that it is possible to provide several function names with the same pragma. It is equivalent to several **`pragma`** lines.

Two similar pragmas are provided that can be used within the scope of a function:

- **`#pragma inline_function (foo,...)`** forces inlining of function **`foo`** every time it is called until the end of the current function,
- **`#pragma noinline_function (foo,...)`** prevents inlining of function **`foo`** every time it is called until the end of the current function.

The two call site scope pragmas take precedence over these two function scope pragmas.

Two lower priority pragma are provided, with file scope:

- **`#pragma inline_file (foo,...)`** to force inlining of function **`foo`** every time it is called until the end of the current source file,
- **`#pragma noinline_file (foo,...)`** to prevent inlining of function **`foo`** every time it is called until the end of the current file.

Finally, to revert inlining policy to the default one (that is, rely on the inliner's evaluation of callee weight), the following pragma is introduced:

```
#pragma defaultinline (foo,...)
```

5.2.4.3 Function naming

As a special case, if the user does not provide any function name, the corresponding pragma applies to all functions called in the scope of the pragma. In this case, parentheses around the function names are optional.

5.2.4.4 User diagnostics

Several warning messages are provided to the user to help track errors.

If two conflicting pragmas are provided only the later is taken into account. For instance,

```
#pragma inline_next (foo)
#pragma noinline_next (foo)
          foo();
```

leads to:

```
warning: #pragma noinline_next (foo) overrides previous #pragma
inline_next (foo)
```

If pragmas are provided at an invalid scope (that is outside of a function), the following message is displayed:

```
warning: #pragma noinline_function (foo) ignored (incorrect scope)
```

To help track misspelling, a warning is also displayed if a pragma could not be applied to any function call:

```
#pragma noinline_next (bar)
     foo(i);
```

leads to:

```
warning: #pragma noinline_next (bar) matched no call
```

5.2.4.5 noinline and always_inline attributes

In order to enable the user to inhibit inlining of one function wherever it is called, the **noinline** attribute is introduced, and is used at the function declaration level.

Conversely, to enable the user to force inlining of one function wherever it is called, the **always_inline** attribute is introduced.

5.2.4.6 Precedence

Command-line options **-INLINE:must=foo** and **-INLINE:never=foo** take precedence over both pragmas and attributes.

Attributes take precedence over pragmas. That is, a function declared with **__attribute__((noinline))** is never inlined, regardless of **pragma inline_*xxx*** statements. However, the user can override this behavior with the **-INLINE:must=foo** command-line option.

If several contradictory pragmas with same scope apply to the same function, the last one overrides the other ones.

5.2.4.7 Examples

Example 1

This example illustrates the use of the **#pragma noinline_next** directive. All calls to **f1()** are candidates for inlining, except the one directly following **#pragma noinline_next**.

```
int ig = 0;
inline void f1(int i) {ig += i;}
void main()
{
   f1(1);         // f1 is candidate for inlining
#pragma noinline_next (f1)
   f1(2);         // f1 is not marked for inlining
   f1(3);         // f1 is candidate for inlining
   printf("result is %d\n", ig);
}
```

Example 2

This example illustrates the use of the **#pragma inline_function** directive. All calls to **f1()** following the **#pragma inline_function (f1)** directive are forced to be inlined, except the one directly following **#pragma noinline_next (f1)**. The call to **f2()** following the **#pragma inline_next (f2)** is also forced to be inlined, while the first call to **f2()** is only a candidate for inlining (inlining depends on the respective weights of **f2()** and its caller).

```
int ig = 0;
int jg = 0;
inline void f1(int i) {ig += i ;}
inline void f2(int i) {jg += i ;}
void main()
{
#pragma inline_function (f1)
   f1(1);         // f1 is forced to be inlined
   f2(1);         // f2 is candidate to inlining
#pragma noinline_next (f1)
   f1(2);         // f1 is not marked for inlining
#pragma inline_next (f2)
   f2(3);         // f2 is forced to be inlined
   f1(3);         // f1 is forced to be inlined
   printf("result is %d %d\n", ig, jg);
}
```

Example 3

This example illustrates the use of the **#pragma defaultinline** directive.

```
int ig = 0;
int jg = 0;
inline void f1(int i) {ig += i ;}
inline void f2(int i) {jg += j ;}
void main()
{
#pragma noinline_function (f1)
   f1(1);         // f1 is not marked for inlining
   f2(1);         // f2 is candidate to inlining
#pragma inline_next (f1)
   f1(2);         // f1 is forced to be inlined
#pragma noinline_next (f2)
   f2(3);         // f2 is not marked for inlining
#pragma defaultinline (f1)
   f1(4);         // f1 is candidate to inlining
   printf("result is %d %d\n", ig, jg );
}
```

Example 4

This example illustrates the use of several function names or an empty name list with **#pragma** directives.

```
#pragma noinline_file ()
int f(int i) { return i+1; }
int g(int i) {
     int j=i+f(i);          // f is not marked for inlining
#pragma inline_next (f,g)
     j += f(i);            // f is forced to be inlined, g is ignored
     j += f(i) + f(i);     // f is not marked for inlining
     return j;
}
int h(int i) {
#pragma noinline_next ()
     int j=i + f(i) + g(i);    // f and g are not marked for inlining
#pragma inline_next (f,g)
     j+=i + f(i) + g(i);       // f and g are forced to be inlined
     return j;
}
```

```
void main()
{
   // g and h are not marked for inlining
   printf("result is %d %d\n", g(0), h(0));
}
```

Example 5

This example illustrates the use of the **noinline** attribute and shows how the attribute has precedence over **#pragma**.

```
#pragma inline_file(f3)
int ig = 0;
void  __attribute__ ((noinline)) f3(int i) { ig += i ; }
int main()
{
   f3(1);         // f3 is not marked for inlining
#pragma inline_next(f3)
   f3(2);         // f3 is not marked for inlining
#pragma defaultinline (f3)
   f3(3);         // f3 is not marked for inlining
   printf("result is %d\n", ig);
}
```

5.3 Memory dependences in C programs

Precise memory dependencies analysis is key to compilation optimization, since it enables the compiler to more freely schedule load instructions above store instructions. By default, a C compiler assumes that any pair of memory accesses which reference distinct types are not aliased (that is, memory dependent). However, real world cases almost always involves pointers to the same types that are actually un-aliased: the compiler cannot generally deduce this property and must rely on additional information. This effect can be achieved either through the C language **restrict** keyword, or with compiler option:

-OPT:alias=*value*

where possible values are listed in *Table 28*.

value	**Description**
any	The default. Any pair of memory accesses may be aliased.
typed	Any pair of memory accesses that reference distinct types are not aliased.
unnamed	Assume pointers never point to global objects.
restrict	Assume that different pointers never point to the same area
disjoint	Assume multiple pointer indirection never overlap.

Table 28: Possible value to the -OPT:alias option

Although the compiler is able to compute precise memory dependences in many cases, this is not possible when complex memory accesses are involved, such as in the following example:

```
for (i = 1; i < n; i ++) {
    a[i-1] = a[i] + b[i];
}
for (i = 1; i < n; i ++) {
    c[d[i]] = c[i] + 1;
}
```

On the first loop, the compiler can fully determine the dependences between memory accesses, provided that it knows that **a** and **b** point to distinct memory locations (see the C language **restrict** qualifier). On the second loop however, without information on values in **d**, the compiler assumes that all memory accesses in the loop are dependent. In particular, the sequence of load and store memory accesses in the iterations of the loop must be strictly respected, resulting in a poor instruction schedule if the loop is unrolled or software pipelined.

A useful property for loop optimizations is when a loop is vectorizable. This property can be enforced on a loop by using the **#pragma loopdep VECTOR**. A vectorizable loop is such that it can be decomposed into a sequence of loops, one per statement of the original loop, without changing the program results. Moreover, for each loop resulting from that decomposition (that contains only one statement), all load memory accesses can be performed before all store memory accesses, which means that a vector version of the loop can be written. In practice, unless the target processor is a real vector processor, the compiler does not decompose vectorizable loops as described. Rather, it uses the vectorizable property of the original loop to remove dependences between memory accesses.

In the example above, the first loop is vectorizable, provided that **a** and **b** do not overlap. The second loop is also vectorizable if the assertion (**d[i]<=i**) holds for all **i**.

Another useful property for loop optimizations is when a loop is parallelizable. This property can be enforced on a loop by using the **#pragma loopdep PARALLEL**. A parallelizable loop is one for which, memory accesses that reference a given memory location may occur only in the same iteration of the loop. As a result, the sequence of memory accesses of the original loop can be changed in any way that preserves the relative order of memory accesses originating from the same loop iteration. Note that a parallelizable loop is always vectorizable, so the **#pragma loopdep PARALLEL** is stronger (but less generally applicable) than the **#pragma loopdep VECTOR**.

In the example above, the first loop is not parallelizable. The second loop is parallelizable if the assertion (**d[i]==i**) holds for all **i**.

The last useful property for loop optimizations is when a loop is liberal. This property can be enforced on a loop by using the **#pragma loopdep LIBERAL**. A liberal loop is one where all its memory accesses reference unique memory locations. As a result, all the memory accesses in the loop can be freely reordered. Note that a liberal loop is always parallelizable, so the **#pragma loopdep LIBERAL** is stronger (but less generally applicable) than the **#pragma loopdep PARALLEL**.

In the example above, the second loop is liberal if the assertion: (**d[i]<1 || d[i]>=n**) holds for all **i**. (For clarity, we omitted this case for the **VECTOR** and **PARALLEL** pragmas.)

The **restrict** qualifier, which applies to pointers or arrays in a C program, is also highly useful to remove dependences between memory accesses inside and outside loops. The restrict property states that two memory accesses originating from different pointers or arrays cannot reference the same memory location, when at least one of the pointers or array has the restrict qualifier. Please note that all memory accesses based on a given restrict pointer or array are still assumed dependent, unless it is obvious to the compiler that they are not, or there is a **#pragma loopdep** on the loop that applies to these dependences.

5.4 Aliasing rules in C/C++ programs

The **-fstrict-aliasing** option allows the compiler to assume the strictest aliasing rules applicable to the language being compiled (the aliasing rules are stated in clause 6.5 (7) of the ISO/IEC Standard (Expressions)).

For C and C++, this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an **unsigned int** can alias an **int**, but not a **void*** or a **double**. A character type may alias any other type.

By default, the **-fno-strict-aliasing** option is used, for legacy reasons. However, in future releases, the **-fstrict-aliasing** will be used by default for **-O** optimization levels.

Note: Particular attention is required before reporting any compiler issue when using **-fstrict-aliasing***, specifically when code runs correctly with the default compiler, but diverges when* **-fstrict-aliasing** *is used. This is often caused by a violation of aliasing rules, which are part of the ISO C/C++ standard. These rules say that a program is invalid if you try to access a variable through a pointer of an incompatible type.*

The example shown in *Figure 2* demonstrates this violation, where a **float** is accessed through a pointer to integer:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  float a = 0.0f ;
  int *pa = (int *)&a ;
  *pa = 0x40000000; /* violation of aliasing rules */
  if (a != 0.0f)
      puts("LEGACY BEHAVIOUR") ;
  else
      puts("STRICT ALIASING BEHAVIOUR") ;
  return 0;
}
```

Figure 2: Aliasing example source

The aliasing rules were designed to allow compilers to perform more aggressive optimization. Basically, a compiler can assume that all changes to variables happen through pointers or references to variables of a type compatible to the accessed variable. De-referencing a pointer that violates the aliasing rules results in undefined behavior.

In the case above, the compiler may assume that no access through an integer pointer can change the float **a**. Thus, the actual value of **a** may be unaffected by the writing through **pa**. What really happens is up to the compiler and may change with architecture and optimization level.

To disable optimizations based on alias-analysis for 'faulty legacy code', the option **-fno-strict-aliasing** must be used as a work-around.

Note: Because the practice of reading from a different union member other than the one most recently written to (called "type-punning") is common, even with **-fstrict-aliasing***, type-punning is allowed, provided the memory is accessed through the union type.*

So, to fix the code above, you can use a union instead of a cast, as shown in *Figure 3 on page 73*.

Note: This is a GCC extension which might not work with other compilers.

```
#include <stdio.h>
/*
  According to GNU documentation, this code should work in
  both strict and non-strict aliasing rules
 */

int main(int argc, char *argv[])
{
union {
     float f ;
     int i;
} u;

     u.f = 0.0f ;
     u.i = 0x40000000 ; /* is 2.0f */
     if (u.f != 2.0f)
          puts("NON-GNU BEHAVIOUR") ;
     else
          puts("GNU ALIASING BEHAVIOUR”) ;
  return 0;
}
```

Figure 3: Aliasing example, using a union

Now the result is always "**GNU ALIASING BEHAVIOUR**".

Finally, to fully respect the ANSI C/C++ aliasing rules, it is necessary to write the data through a character type before reading it again, see *Figure 4 on page 74* and *Figure 5 on page 75*.

The drawback of this standard conforming solution is that it has to account for endianness, and that it is less efficient than simply writing through an integer:

```
#include <stdio.h>

/*
  According to ANSI standard, this code should work in
  both strict and non-strict aliasing rules
 */

#include <stdio.h>

#define EXTRACTBYTE(val, pos) (((val) >> (pos*8)) & 0xff)

int main(int argc, char *argv[])
{
  union
  {
    float f ;
    char c[4] ;
  } u;

  const unsigned int twoasint = 0x40000000 ;

  u.f = 0.0f ;

#if defined(__BIG_ENDIAN__)
  u.c[0] = EXTRACTBYTE(twoasint, 3) ;
  u.c[1] = EXTRACTBYTE(twoasint, 2) ;
  u.c[2] = EXTRACTBYTE(twoasint, 1) ;
  u.c[3] = EXTRACTBYTE(twoasint, 0) ;
#elif defined(__LITTLE_ENDIAN__)
  u.c[0] = EXTRACTBYTE(twoasint, 0) ;
  u.c[1] = EXTRACTBYTE(twoasint, 1) ;
  u.c[2] = EXTRACTBYTE(twoasint, 2) ;
  u.c[3] = EXTRACTBYTE(twoasint, 3) ;
```

Figure 4: Aliasing example, writing through a character type, page 1 of 2

```
#else
#error "Unknown endianness : please define either
__BIG_ENDIAN__ or __LITTLE_ENDIAN__"
#endif

  if (u.f != 2.0f)
      puts("UNEXPECTED BEHAVIOUR") ;
  else
      puts("ANSI ALIASING BEHAVIOUR") ;

  return 0;
}
```

Figure 5: Aliasing example, writing through a character type, page 2 of 2

In this case, the program always prints "**`ANSI ALIASING BEHAVIOR`**" regardless of the compiler and its optimization options.

5.5 Data prefetching

The ST200 processor includes a data prefetching mechanism through a **PFT** instruction. Prefetching data ahead of its actual use, enables some of the data cache miss penalties to be hidden. Data cache miss cycles often account for a significant fraction of execution cycles.

The **st200cc/st200c++** compiler includes support for user and automatic data prefetch instructions:

- User data prefetch instructions are inserted in the C/C++ code using a **`__builtin__prefetch`** builtin.
- Automatic data prefetch is generated by the compiler at optimization level **`-O3`**, using specific command line options described in the following sections.

5.5.1 Manual data prefetching

```
void __builtin_prefetch (const void *addr)
```

You can insert calls to **__builtin_prefetch** into code, for which you know addresses of data in memory are likely to be accessed soon, in order to generate a data prefetch instruction.

The value of **addr** is the address of the memory to prefetch.

Data prefetch does not generate faults if **addr** is invalid, but the address expression itself must be valid. For example, a prefetch of **p->next** does not fault if **p->next** is not a valid address, but evaluation does fault if **p** is not a valid address.

The following option controls whether the **__builtin_prefetch** is accounted for, this can be useful to measure the effect of the insertion of the builtin:

The **-m[no-]builtin-prefetch** option disables or enables the **__builtin_prefetch** builtin throughout the compilation. For example, with **-mno-builtin_prefetch** the compiler allows calls to **__builtin_prefetch** in the source code, however, they are ignored.

5.5.2 Automatic data prefetching

Automatic prefetching may be enabled by the command line option:

-m[no-]auto-prefetch

Note: 1 This option is only effective when optimization level **-O3** *is selected.*

2 Automatic prefetching is disabled by default.

Automatic prefetching inserts prefetch instructions for array references inside loops. In order to generate a prefetch for a memory reference the compiler may need to generate the prefetch on an earlier iteration of the loop to the memory reference. To calculate the address that is used in the later iteration the compiler requires the difference between the load address and the prefetch address to be an integer constant.

For example, given the following code:

```
for (i=0; i < 10; i++) {
....
load array [i]
}
```

In *Figure 6*, on the **i**th iteration we want to load array **[i]** and eight iterations later we will want to load array **[i+8]**. To speed up the load of array **[i+8]** the compiler prefetches array **[i+8]** while it loads array **[i]**. This is possible because array **[i+8]** is a known constant distance from array **[i]**.

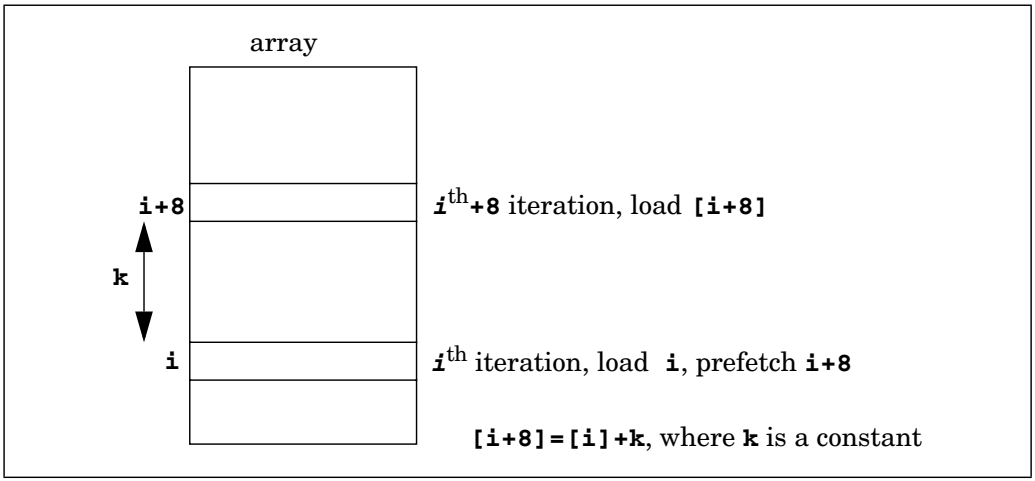

Figure 6: Pre-fetching on earlier iteration of loop

The resulting compiled code has the prefetch instruction added:

```
for (i=0; i < 10; i++) {
....
load array [i]
prefetch array [i+8]
}
```

The compiler can handle more complex cases than this but in all cases the distance between the load address and the prefetch address (distance **k** in *Figure 6*) must be a known constant[1].

1. Mathematically the load address must be a linear function of the enclosing loop iteration variables.

The following code gives a further example of prefetching:

```
extern int A[160];
int foo(void)
{
     int i;
     int sum = 0;
     for (i = 0; i < 160; i++)
          sum += A[i];
     return sum;
}
```

The compiler's loop unrolling optimization first unrolls this by a factor of 8 to give:

```
extern int A[160];
int foo(void)
{
     int i;
     int sum = 0;
     for (i = 0; i < 160; i+=8) {
          sum += A[i];
          sum += A[i+1];
          sum += A[i+2];
          sum += A[i+3];
          sum += A[i+4];
          sum += A[i+5];
          sum += A[i+6];
          sum += A[i+7];
     }
     return sum;
}
```

After the unrolling, the compiler inserts prefetch operations to give code equivalent to:

```
extern int A[160];
int foo(void)
{
     int i;
     int sum = 0;
     for (i = 0; i < 160; i+=8) {
          sum += A[i];
          sum += A[i+1];
          sum += A[i+2];
```

```
            sum += A[i+3];
            sum += A[i+4];
            sum += A[i+5];
            sum += A[i+6];
            sum += A[i+7];
            __builtin_prefetch(&A[i+16]);
        }
        return sum;
}
```

5.5.3 Advanced data prefetching options

By default, the prefetch instructions inserted read two cache lines ahead of the current array reference. This behavior may be adjusted by using the command line option:

-LNO:prefetch_ahead=*n*

where **n** is the number of cache lines to read ahead. For example:

-LNO:prefetch_ahead={0,50}[2]

where **{0,50}[2]** means that the **-LNO:prefetch_ahead** option expects an integer argument whose range is in the **[0,50]** interval, for which the default value is **2**.

Alternatively, it is possible to specify the number of loop iterations to prefetch ahead using the command line option:

-**LNO:prefetch_iters_ahead=*n***.

which makes the compiler prefetch array references for the loop ***n*** iterations after the current iteration.

-**LNO:prefetch_iters_ahead={0,50}[2]**

Caution: The compiler may also automatically unroll the loop: the value of ***n*** specifies the number of loop iterations before this automatic unrolling.

Automatic prefetching is not performed upon arrays that have the built-in function **__builtin_prefetch** applied to them.

For example, the option:

-LNO:prefetch_iters_ahead={0,50}[2]

controls how many iterations an automatic prefetch instruction must be issued ahead of its associated memory reference. This value is used only if the step of the memory access in an iteration of the loop is greater or equal to the size of a data cache line.

The option **-LNO:clean_miss_penalty={0,1000000000}[140]** controls the number of cycles between the issue of a prefetch instruction and the availability of the data in the prefetch buffer.

The option **-LNO:dcache_prefetch_buffers={0,1000}[8]** controls the number of prefetch buffers available on the processor. These prefetch buffers are used to store prefetched data before a memory access move it to the data cache.

5.6 Profiling

Before optimizing any application, it is recommended that you analyze the critical areas of your code.

Profiling creates an instrumented program from your source code. Each time this instrumented code is executed, the instrumented program generates an information file that can be later displayed with the **st200gprof** tool.

This section is not a complete guide to profiling, but a quick refresher on how to proceed with the compiler. More details about using **gprof** can be found at http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html.

Example:

```
#st200cc -O2 -pg *.c -o myexe
```

After first run, a file **gmon.out.000** is generated. It can be viewed with **st200gprof** with the following command:

```
#st200gprof myexe gmon.out.000
```

After each run in the same directory, the file name suffix part of the **gmon.out.000** is incremented, so that profile information for the new run is available as **gmon.out.001**, and so on.

The symbolic information available in the profile information can be augmented by using the **st200cc -g** option when compiling the source code.

5.7 Call trace instrumentation

This section describes the use of the **-finstrument-functions** and **-minstrument-calls** options.

5.7.1 Instrumenting functions: -finstrument-functions

The **-finstrument-functions** option provides standard GCC functionality. Using this option generates instrumentation calls for entry and exit to functions.

Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site:

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

- The first argument is the address of the start of the current function, which may be looked up specifically in the symbol table.
- The second argument is the address of the call site from where the current function was invoked. It corresponds to an address in the range of the caller function addresses that may be found in the symbol table of the executable.

The functions that are inlined by the compiler are not instrumented.

To force instrumentation of all functions use the **-fno-inline** option to disable inlining.

A function may be given the attribute **no_instrument_function**, in which case this instrumentation is not done. This can be used, for example, for the profiling functions listed above, high-priority interrupt routines, and any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory).

The program must be linked with an object file that implements the two functions above to link correctly.

5.7.2 Instrumenting call to functions: -minstrument-calls

The **-minstrument-calls** is not a standard GCC option.

Using this option generates instrumentation calls just before, and just after each function call.

The following profiling function is called with the address of the caller function and the address of the callee function:

```
void __profile_call (   void *caller_fn,
                        void *callee_fn,
                        const char *caller_name,
                        const char *callee_name,
                        int event);
```

- The first argument is the address of the start of the current function (the caller function), which may be looked up specifically in the symbol table.
- The second argument is the address of the start of the called function (the callee function), which may be looked up specifically in the symbol table.
- The third argument is the name of the caller function.
- The fourth argument is the name of the callee function, or **NULL** if the call is an indirect call.

  The function names passed in the third and fourth arguments are pointers to static strings that have the lifetime of the instrumented executable or shared object.

  The function names are the mangled names in C++.
- The fifth argument is **0** when this function is invoked just before a call, instrumenting a function entry. It is **1** when this function is invoked just after a call, instrumenting a function exit.

Function calls that are inlined by the compiler are not instrumented.

To force instrumentation of all functions use the **-fno-inline** option to disable inlining.

A function may be given the attribute **no_instrument_function**, in which case this instrumentation is not done if the caller or the callee function has the attribute **no_instrument_function**.

The program must be linked with an object file that implements the function above to link correctly.

The main differences with the **-finstrument-functions** option are:

- this instrumentation tracks (caller, callee) address pairs instead of (call_site, callee) address pairs. If the call site information is required use the **-finstrument-functions** option,
- this instumentation provides the caller and callee name when available, which avoids a specific post processing pass to retrieve the function names,
- this instrumentation is at the call site and not in the callee, thus for instance calls to top level library functions (which are not instrumented) are seen while **-finstrument-functions** do not see them, to disable the instrumentation of the call to a particular library routine you must declare it with the **no_instrument_function** attribute,
- this instrumentation is not standard GCC functionality.

5.8 Profiling feedback optimization (PFO)

5.8.1 Principles

The **st200cc** profiling feedback model is a three step process.

- Instrumentation of C/C++ sources.
- Execution of the binary on sample input, generating files of feedback data. The run can be executed either on a simulator or on hardware.
- Feedback annotation of C/C++ at the same point as in the first compilation. Sources are enhanced with feedback data that are used and refined during the compilation process.

st200cc feedback instruments the input sources early in the compiler back-end. Consequently, feedback annotations are almost independent from optimization levels.

5.8.2 Command line

The **st200cc** command line option **-fb_create <*name*>** is used to generate an instrumented executable program. This can be used to produce one or more ***name*.instr0.#####** files for subsequent feedback compilation.

When the **-c** option is used to produce an object file (that is eventually linked to form an instrumented executable), the **-fb_create** option should also be specified at link time. If different names are specified, the one used to compile the main file (where the **main** procedure is) will be used.

The executable is instrumented with special instructions that generate information that the compiler can use to better optimize your program.

When this instrumented executable is run (usually with a representative training input set) that information is stored in a unique file called ***name*.instr0.#####**. An unique file is generated for each execution.

The **st200cc** command line option to specify the feedback files to be used is **-fb <name>**. All files with the name ***name*.instr0.######** will be combined.

5.8.3 Example

Step 1

Instrument the program to set it up for running a test set with profiling tests enabled. To do this, compile the program with the **-fb_create *name***. This option inserts code into the program so it can track which spots of the code are used often when the program is run (for example, how many times a branch is taken or how many times a loop iterates). The library **libinstrC.a** is automatically linked with the program.

```
$ st200cc -fb_create fbtest -o ftest ftest.c
```

Step 2

Run the program with some sample input. When the program executes, statistics are generated and written to files for later use. The instrumented program runs significantly slower than usual. This creates a file of frequency data, named **fbtest.instr0.######**. The **0** in **instr0** indicates that instrumentation occurred before Very High Optimization (VHO) lowering which is the very first optimization of the back end compilation process. Eventually, it should be possible to instrument at a variety of stages, but this is not activated for the moment. Multiple runs can be performed using different input data to generate multiple frequency data files. Runs can be executed either on simulators or hardware, the only restriction is to have access to the host file system.

```
$ st200run -tsim -- ./ftest
```

Step 3

Re-compile the program with the **-fb *name*** option. This tells the compiler that it has some sample statistics from which it can optimize code better. It then uses these statistics to compile a new program that is better tuned for the input. Feedback data will be read and combined from all files with names **fbtest.instr0.######**.

Note: It is necessary to delete obsolete feedback data files (for instance if the code is changed) or use a different feedback file name (for instance to evaluate enhancements on various data sets).

```
$ st200cc -fb fbtest -o ftest ftest.c
```

5.8.4 What feedback does

Feedback collects frequency counts on control flow such as branches, loops, and procedure calls. Memory accesses are not counted.

During the instrumentation step, the compiler inserts extra code into the original code. The inserted code consists of procedure calls to **libinstrC.a** routines which gather control flow data, such as the number of times a branch is taken or not taken, how many times a loop iterates, or how many times procedures are invoked.

When the instrumented binary is invoked on the test data, the program performs as normal, except that the **libinstrC.a** routines collect control flow frequencies and store the counts into a file when the program completes.

During the annotation step, the compiler reads the frequency data from any available feedback data files and attaches the data to the code. Annotation occurs at the same point of compilation that instrumentation occurred.

Later phases of the compiler (for instance instruction cache optimization, if-conversion and code generation) can use the data to guide optimization decisions.

5.9 Interprocedural analysis optimization (IPA)

5.9.1 Introduction

The **-ipa** option enables interprocedural analysis. The compiler identifies optimization opportunities across module boundaries. It does this by extending the scope that is examined during optimization and inlining from a single module to multiple modules.

The -**ipa** option must be included in both the compile and link phases.

The major benefits of IPA are:

- interprocedural constant propagation,
- interprocedural alias analysis,
- inter-module inlining.

A more advanced use of IPA is function specialization (cloning).

5.9.2 Using IPA

The only necessary option to trigger the IPA compilation is **-ipa**.

No fundamental Makefile changes are necessary to the usual build process, as the compiler driver takes care of all the necessary phases.

The compilation time and the link time is longer as much of the optimization work is driven from the linker. This can be observed by using the **-v** compiler option.

The following steps are performed when building an executable in IPA mode:

- the **.c** files are translated into **special .o** files,
- the **.o** files are merged together (code, symbol table),
- the **.o** files are analyzed and optimized,
- the final link is performed.

5.9.3 IPA command line options

The following table describes advanced **IPA** options.

Option	Description
-IPA:aggr_cprop=ON|OFF	Enables or disables aggressive inter-procedural constant propagation. Attempts to avoid passing constant parameters, replacing the corresponding formal parameters by the constant values. The default in **ON**.
-IPA:cgi=ON|OFF	Enables or disables constant global variable identification. This options marks non-scalar global variables which are never modified as constant, and propagates their constant values to all files. The default is **ON**.
-IPA:cprop=ON|OFF	Enables or disables inter-procedural constant propagation. This option identifies formal parameters which always have a specific constant value. The default is **ON**. See also **-IPA:aggr_cprop**.
-IPA:depth=*n*	This option is identical to **maxdepth=*n***
-IPA:dfe=ON|OFF	Enables or disables dead function elimination. This option removes subprograms which are never called from the program. The default is **ON**.
-IPA:dve=ON|OFF	Enables or disables dead variable elimination. This option removes variables which are never referenced from the program. The default is **ON**.
-IPA:forcedepth=*n*	Sets inline depths. Instead of the default inlining heuristics, this option directs IPA to attempt to inline all functions at a depth of (at most) ***n*** in the call graph, where functions which make no call are at depth 0, those which call only depth 0 function are at depth 1, and so on. This ignores the default heuristic limits on inlining.
-IPA:inline=ON|OFF	Performs inter-file subprogram inlining during main IPA processing. The default in **ON**.
-IPA:keeplight=ON|OFF	Directs IPA to not send **-keep** to the compiler, in order to save space. The default is **ON**. Setting it to **OFF** will leave intermediate files in a directory which has the name of the final executable suffixed with **.ipakeep**.

Table 29: Advanced IPA options

Option	Description
-IPA:maxdepth=*n*	Directs IPA to not attempt to inline functions at a depth of more than ***n*** in the call graph, where functions which make no call are at depth 0, those which call only depth 0 functions are at depth 1, and so on. Inlining remains subject to overriding limits on code expansion. See also **forcedepth**, **space** and **plimit**.
-IPA:multi_clone=*n*	Specifies the maximum number of clones that can be created from a single procedure. By default, this value is 0. Aggressive procedure cloning may provide opportunities for inter-procedural optimization, but it also may significantly increase the code size.
-IPA:node_bloat=*n*	When used in conjunction with **-IPA:multi_clone**, this specifies the maximum percentage growth of the total number of procedures relative to the original program.
-IPA:plimit=*n*	Stops inlining into a particular subprogram once it reaches size ***n*** in the intermediate representation. The default is 2500.
-IPA:space=*n*	Stops inlining when the program size has increased by ***n***%. For example, ***n***=20 limits code expansion due to inlining to approximately 20%. The default is 100%.
-IPA:specfile=filename	Opens filename to read more options. A spec file contains zero or more of the options allowed by IPA.

Table 29: Advanced IPA options

5.9.4 Limitations

The IPA optimization is not compatible with the **-g** compiler option.

The IPA optimization is currently not supported with the PFO optimization (see *Section 5.8 on page 83*),

5.10 Symbol visibility specification

Command line options are provided to control symbol visibility.

These options are useful in the context of position independent code generation(**-fpic**), where symbol visibility has a significant impact on code performance and the dynamic symbol table size.

Two options are provided:

- **-fvisibility=[default|protected|hidden|internal]**: this option specifies the default visibility to apply to symbols defined in the current source file.

Note: This does not set the visibility of symbols declared externally.

- **-mvisibility-decl=<*file*>** : this option specifies a visibility specification file that the compiler can use to determine the visibility for a symbol declaration or defintion.

These options are useful to optimize a shared object DSO or a relocatable library as defined by the ST200 relocatable library model described in the *ST200 Cross Development Manual*.

5.10.1 Introduction

In the context of **-fpic** code generation, symbol visibility has a significant impact on code generation.

In addition to GP relative addressing which is an invariant of PIC code, some other code generation overheads appear depending on the symbol visibility.

These overheads are summarized in *Table 30*:

Overhead	Default visibility	Protected visibility	Hidden visibility	Internal visibility
The GP value must be set up on entry to the function symbol as the function can be called from outside, this generates a specific sequence in the prologue of the function.	Yes	Yes	Yes	No[f]
A pointer to the symbol (function or data) is taken from the GOT entry, generating a load instead of a single address computation and some additional data for the GOT entry.	Yes	Yes	No[c]	No[c]
Inlining of the function symbol is not possible as the symbol is preemptible	Yes	No[b]	No[b]	No[b]
A call to the function symbol goes through a stub function in the PLT, generating one additional jump to the stub function and some additional code for the stub function.	Yes[a]	Yes	No[d]	No[d]
The symbol appears in the dynamic symbol table, this augments the size of the loadable data segment.	Yes[a]	Yes	No[e]	No[e]

Table 30: Visibility overheads

a. This overhead can be removed by some link time options and by the use of linker version scripts. These link time options are not used by the compiler to optimize the code.

b. Inlining of the function symbol is possible as the symbol is not preemptible.

c. A pointer to the symbol can be built from a GP relative address computation as the symbol is local to the library.

d. A call to the function symbol is direct as the symbol is local to the library.

e. The symbol does not appear in the dynamic symbol table as it is local to the library.

f. There is no need to set up the GP value as the function was called from another function inside the same library, and thus the GP value is already set.

Internal visibility has no remaining overheads.

To support visibility declarations the current **st200cc** compiler implements:

- visibility attribute GNU extension[1].
- linker option **-Bsymbolic** which has the same effect as protected visibility but at link time only.
- linker version scripts that allow the user to explicitly list the symbol that is required to be exported globally, thus reducing the dynamic symbol table size.

The following sections give example usage of the **-fvisibility** and **-mvisibility-decl** options.

5.10.2 Usage of the -fvisibility option

This option is mainly used to declare by default all symbols as having hidden visibility.

The advantage of this is that the user has to declare by using attributes the functions or data that he explicitly wants to be global.

For instance if the interface for a library is reduced to three functions (initialize, process and finalize) the user can enable better code generation by including the **config.h** file (see *Figure 7*) in its sources and using the **-fvisibility=hidden** option at compile time.

```
#define VISIBILITY(v) __attribute__((visibility(#v)))
extern void initialize(void) VISIBILITY(protected);
extern void process(void) VISIBILITY(protected);
extern void finalize(void) VISIBILITY(protected);
```

Figure 7: config.h

1.The **__attribute__((visibility(...)))** GNU extension does not apply to static symbols as ELF defines visibility only for global symbols. This is unfortunate as the compiler can take advantage of the fact that a static function is internal. Though the extensions allow static symbols to be declared internal.

This informs the compiler that all functions or data defined for the compilation of the library are hidden, except those explicitly declared protected in the **config.h** file.

Without the **-fvisibility** option the user would have to attribute all of the functions definition with the **VISIBILITY(hidden)** macro.

Note: The **-fvisibility=hidden** *option has some limitations. It applies only to function definitions, not to declarations. Thus, if a file that is part of the library references a symbol in another file in the same library, the compiler still generates GOT accesses, as it does not have the information that the definition is effectively in the library.*

To remove the above limitation, the user must use the **-ipa** option that enables the compiler to consider all files at once and thus effectively apply the code generation optimizations.

5.10.3 Usage of the -mvisibility-decl option

The main usage of the **-mvisibility-decl** option is to avoid modifying the source code by adding explicit visibility attributes to function and data, declarations and definitions.

For instance, this effect with the **-fvisibility=hidden** option can be achieved without declaring the visibility attribute in the header file. The file **mylibrary.v** (see *Figure 8*) can replace the **config.h** file. The user then compiles with the options: -**mvisibility-decl=mylibrary.v -fvisibility=hidden**.

```
# Visibility specification for mylibrary
{
protected:
     # symbols in the interface
     initialize;
     process;
     finalize;
}
```

Figure 8: mylibrary.v

Another more advanced usage is to explicitly define all the symbols that are outside of the library, all the symbols that are part of the library interface and let all other symbols be local to the library.

For instance, if the user knows the list of symbols that are used by the library but are external to the library (for example, **printf** and **malloc**) the specification file **mlibrary_all.v** shown in *Figure 9* can be used to compile with the command line option: **-mvisibility-decl=mlibrary_all.v**.

```
# Visibility specification for mylibrary
{
default:
     # symbols usable outside of the library
     printf;
     malloc;
protected:
     # symbols in the interface
     initialize;
     process;
     finalize;
hidden:
     # all other symbols are kept local
     *;
}
```

Figure 9: mylibrary_all.v

With the **mylibrary_all.v** visibility specification the library designer asserts the following:

1 No symbol other than (**printf**, **malloc**) can be referenced from the library without being defined in the library. If this assertion if violated a static link-time error will occur.

2 Only the symbols (**initialize**, **process**, **finalize**) are accessible to the other modules. There will be a dynamic link time error if this assertion is violated by another module.

3 All other symbols are local to the library.

The advantage of 1 is that the library dependencies on other modules are explicit. It can be useful for libraries, though in some contexts it may be time consuming to enumerate all such symbols.

The advantage of 2 is that the library interface is explicit and an error occurs if it is violated.

The advantage of 3 is that the compiler can optimize all these hidden symbols and optimize all the references to these hidden symbols.

Note: 1 The **-ipa** *option is not necessary in this case to optimize reference to symbols as opposed to the* **-fvisibility=hidden** *option.*

2 The **-ipa** *option is still useful to further optimize the library by identifying from the set of hidden symbols which ones are internal. A sufficient condition for a symbol to be internal is that it is hidden and the address is never taken. The* **-ipa** *option can compute the 'address taken property' and use the fact that the symbol has been declared hidden to set it internal.*

The visibility specification file can be used to specify internal visibility for some symbols, though it must be done very carefully as in this case no link time (static or dynamic) error occurs if the symbol is passed to another module (thus violating the internal property). This must only be done in cases where the library designer is sure that an address is not passed outside of the library.

5.10.4 The visibility specification file

The visibility specification file resembles the version script files used by the GNU linker except that it specifies all levels of visibility instead of only the global and local properties.

The specification file has the following BNF grammar as shown in *Figure 10*, with the additional notion that the '-' denotes a range (for instance, a-z represents all characters from a to z inclusive):

```
visibility_spec  ::= { { visibility_decls } }
visibility_decls ::= { visibility_decl }
visibility_decl  ::= VISIBILITY : pattern_decls
pattern_decls    ::= { pattern_decl }
pattern_decl     ::= lang_decl | symbol_decl
lang_decl        ::= extern " LANGUAGE " { symbol_decls }
symbol_decls     ::= { symbol_decl }
symbol_decl      ::= PATTERN ;
VISIBILITY       ::= default | protected | hidden | internal
LANGUAGE         ::=  "C" | "c" | "C++" | "c++"
PATTERN          ::= {a-z | A-Z | 0-9 | _ | : | [ | ] | ? | * }
```

Figure 10: Visibility specification file BNF

The ***PATTERN*** is a simplified regular expression suitable for 'globbing' as generally found in shell command line expansion. The implementation uses the 'globbing' function **fnmatch()** of the **libiberty.a** GNU library.

The **LANGUAGE** is one of C or C++ and specifies the demangling scheme that must be used to match a symbol.

If no language declaration is specified the pattern must match the assembly name of the symbol.

For example, *Figure 9* is a commented visibility specification file **dummy.v**:

```
# A sample visibility specification file
{
protected:
     # will match the symbol named my_func
     my_func;
     # will match the symbol named my_data
     my_data;
     # will match all symbols prefixed by protected_
     protected_*;
default:
extern "C++" {
     # will match all C++ symbols of std:: namespace
     std::*;
     # will match all typeinfo information of std:: namespace
     typeinfo*std::*;
     # will match all vtable of std:: namespace
     vtable*std::*;
     # will match all vtable tables of std:: namespace
     VTT*std::*;
}
hidden:
extern C++ {
     # will match all methods, data of the mycomplex class
     mycomplex::*; #
}
     # will match the vtable for complex using the mangled name
     _ZTV9mycomplex;
}
```

Figure 11: dummy.v

6 GNU C extensions supported by st200cc

6.1 Introduction

GNU **cc** provides a large set of extensions that are widely used in the GNU Linux community. These extensions can be used to:

- describe embedded features, for example, data section placement,
- provide guidance to the compiler for optimization, for example, the **`noreturn`** function,
- provide language extensions, for example, conditional **`lvalue`** or C99 features,
- instruct the compiler to modify the Application Binary Interface, for example using the **`packed`** attribute.

The GNU extensions are sometimes the only way to access ELF features that are not directly available in the C language, for example, to declare a symbol weak.

6.2 Extensions to the C language family

st200cc provides several language features not found in ANSI standard C. (The "**-pedantic**" option directs **st200cc** to print a warning message if any of these features are used.) To test for the availability of these features in conditional compilation, check for a predefined macro **__GNUC__**, which is always defined under **st200cc**.

It is recommended to always put code containing **st200cc** extensions under the C pre-processor macro **__GNUC__**.

```
#if __GNUC__
  /* Original GNU code */
#else
  /* Work-around code */
#endif
```

6.2.1 Statements and declarations in expressions

Statements and declarations in expressions allow complicated C statements to be written and used as if they were a simple C expression, optionally returning a result value. Local declarations and labels may be embedded.

This provides a way to construct a safe preprocessor macro that comprises several statements, without using the **do { } while(0)** trick that swallows the semi-colon.

```
#define cfoo() \
  ( { int y = foo (); int z; \
      if (y > 0) z = y;      \
      else z = - y;          \
      z; })
```

6.2.2 Locally declared labels

When GNU extensions are used in conjunction with expression statements and macros, they enable **service labels** to be used, that is labels whose scope is limited to the current statement. For example:

```
#define SEARCH(array, max, target)                \
({                                                \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { value = i; goto found; }                \
  value = -1;                                     \
 found:                                           \
  value;                                          \
})
```

6.2.3 Labels as values

The address of a label defined in the current function, or a containing function, can be obtained with the extended **&&** unary operator that has type ***void****. For example:

```
const char * cgoto(int i)
{
  void *ptr = &&foo;
  static void *array[] = { &&foo, &&bar, &&hack };
  goto *array[i];
 foo:
     return "foo" ;
 bar:
     return "bar" ;
 hack:
     return "hack" ;
}
```

6.2.4 Naming an expression's type

A name can be given to the type of an expression using a typedef declaration with an initializer. To define ***name*** as a type name for the type of ***expression***, do:

```
typedef name = expression;
```

This can be used in conjunction with the statements-within-expressions feature described in *Section 6.2.1*. For example, to define a safe “maximum” macro that operates on any arithmetic type:

```
#define max(a,b) \
    ({typedef _ta = (a), _tb = (b);  \
      _ta _a = (a); _tb _b = (b);    \
      _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for **a** and **b**.

Note: In the future the GNU language may include a new form of declaration syntax that allows the declaration of variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

6.2.5 Referring to a type with typeof

typeof allows you to refer to an object data type by referring to an object of that type. It is particularly useful to write generic and safe macro-definitions, which can then be applied to various primitive types or user-defined data types. Without this extension, it is necessary to define as many specific macros as the number of different types used in calls to the generic macro.

```
#define max(a,b) ({ \
   typeof (a) _a = (a); \
   typeof (b) _b = (b); \
   _a > _b? _a: _b; \
})
```

6.2.6 Generalized L-values

Compound expressions, conditional expressions and casts are allowed as **lvalues** provided their operands are **lvalues**. For example:

```
(a, b) += 5;
```

6.2.7 Conditionals with omitted operands

The middle operand in a conditional expression may be omitted, for example:

```
z = x? : y;
```

6.2.8 Double-word integers

long long support (integer 64-bits) is supported by the ST200 Micro Toolset. It is now also an ISO C99 feature.

```
long long x;
```

6.2.9 Hexadecimal floats

Floating-point numbers are written in hexadecimal format:

```
float f = 0x1.fp3;
```

6.2.10 Specifying a register for a local variable

A register may be specified for a local variable, for example:

```
register long r15 asm (“r15”) = name;
```

6.2.11 Array of length zero

Zero length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable length object.

```
#include <stdio.h>
#include <stdlib.h>

struct line {
    int length;
    char contents[0];
};

struct line *newline( unsigned int this_length)
{
    struct line *thisline = (struct line *)
      malloc (sizeof (struct line) + this_length);
    thisline->length = this_length;
    return thisline ;
}

void delline(struct line *thisline)
{
    free(thisline) ;
}

int main(int argc, char *argv[])
{
    enum { __MAXL = 128 } ;
    enum { __L = 16 } ;
    struct line *lines[__MAXL] ;
    int i ;
    printf("sizeof(line) : %d\n", sizeof(struct line)) ;
    for(i=0; i< __MAXL; i++) {
        lines[i] = newline(__L) ;
    }
    for(i=0; i< __MAXL; i++) {
        delline(lines[i]) ;
    }
    puts("Done.") ;
    return 0 ;
}
```

6.2.12 Array of variable length

An array of variable length is an automatic array defined with a length that is not a constant expression. Also known as a VLA.

```
#include <stdio.h>
#include <stdlib.h>

void sadcat(char *s1, char *s2)
{
    char str[strlen (s1) + strlen (s2) + 1];
    strcpy (str, s1);
    strcat (str, s2);
    printf("%s + %s == %s\n", s1, s2, str) ;
    printf ("sizeof(str) = %d\n", sizeof(str));
}

void tester (int len, char buffer[len][len]) {
  int i=0, j=0;
  char tt[len][len];
  for (i=0; i<len; i++)
    for (j=0; j<len; j++)
      buffer [i][j] = i*j;
  printf ("sizeof(tt) = %d\n", sizeof(tt));
  printf ("sizeof(buffer) = %d\n", sizeof(buffer));
}

char data[10][10];

int main(int argc, char *argv[])
{
    sadcat("Foo", "Bar") ;
    tester (4, data);
    tester (10, data);

    return 0 ;
}
```

6.2.13 Macro with variable number of arguments

This extension enables a macro to be defined that can safely be expanded into a function with a variable number of arguments. These macros are also called CPP **vararg** macros.

For example, the following C program:

```
#define eprintf(format, args...) fprintf (stderr, format, ##args)
  eprintf ("success!\n");
  eprintf ("%s%d: ", input_file_name, line_number);
```

gets expanded into:

```
fprintf ((&__iob[2]), "success\n!");
fprintf ((&__iob[2]), "%s%d: ", input_file_name, line_number);
```

Note: GNU C supports two types of "variable number of arguments" syntax. The ISO C99 format, which uses **__VA_ARGS__** *and the GNU format that uses* **##args***. The ISO C99 format does not support the case where the number of parameters passed as part of the ellipsis is zero. GNU C reuses the ## trick to absorb the comma in this case. For example:*

```
#include <stdio.h>
#define gnu_eprintf(format, args...)  \
      fprintf (stdout, "gnu_eprintf " format, ## args)
#define isoc99_eprintf(format, ...)  \
      fprintf (stdout, "isoc99_eprintf " format, __VA_ARGS__)
#define extended_isoc99_eprintf(format, ...)  \
      fprintf (stdout, "extended_isoc99_eprintf " format, ## __VA_ARGS__)
#define errprintf(args...) \
      gnu_eprintf ("errprintf " "%s\n", ## args)

int main(int argc, char *argv[]) {
  /* Try 1, 2, 3 arguments */
  gnu_eprintf ("One argument: %s. Done.\n", __FILE__);
  gnu_eprintf ("Two arguments: %s:%d. Done.\n", __FILE__, __LINE__);

  isoc99_eprintf ("One argument: %s. Done.\n", __FILE__);
  isoc99_eprintf ("Two arguments: %s:%d. Done.\n", __FILE__, __LINE__);

  extended_isoc99_eprintf ("One argument: %s. Done.\n", __FILE__);
  extended_isoc99_eprintf ("Two arguments: %s:%d. Done.\n", __FILE__,
__LINE__);
  extended_isoc99_eprintf ("Three arguments: %s:%s:%d. Done.\n",
__FUNCTION__, __FILE__, __LINE__);
```

```
  /* The case with no arguments ... */
  gnu_eprintf ("No arguments. Done.\n");
  /* The line below causes a syntax error */
  isoc99_eprintf ("No arguments. Done.\n");
  extended_isoc99_eprintf ("No arguments. Done.\n");

  /* Cascade of macros with variable number of arguments */
  errprintf (__FILE__);
  return 0 ;
}
```

6.2.14 Strings literals with embedded newlines

GNU **cpp** permits string literals to cross multiple lines without escaping the embedded newlines. Each embedded newline is replaced with a single newline character in the resulting string literal, regardless of what form the newline took originally.

The macro definition:

```
#define MESSAGE  \
"Hello,
good brave new World!
"
```

would be written under ISO:

```
#define MESSAGE  \
"Hello,\n" \
"good brave new World!\n"
```

6.2.15 Non-Lvalue arrays may have subscripts

In ISO C99, arrays that are not **lvalues** still decay to pointers, and may be subscripted. However, they may not be modified or used after the next sequence point and the unary operator "**&**" may not be applied to them.

```
struct foo {int a[4];};

struct foo f() {
  static const struct foo f = { 2, 4, 8, 16 };
  return f ;
}

void bar (void)
{
    int i;
    for (i=0; i<4; i++)
      printf ("f().a[%d] == %d\n", i, f().a[i]) ;
}

int main(int argc, char *argv[])
{
  bar ();
  f().a[0] = 15;
  bar ();

  return 0 ;
}
```

6.2.16 Arithmetic on void and function pointers

In GNU C, addition and subtraction are supported by pointers to **void** and by pointers to functions. size used for a **void** or of a function is **1**. **sizeof** is allowed for **void** and for a function: it returns 1.

```
void f0(void) {}
void *p = 0;
void (*pf)(void) = 0;
bar (void) {
  p++;
  pf++;
  printf ("sizeof(void) = %d\n", sizeof(void));
  printf ("sizeof(func) = %d\n", sizeof(f0));
}
```

6.2.17 Non-constant initializers

As in standard C++ and ISO C99, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions. For example:

```
int foo (int f, int g)
{
    int beat_freqs[2] = { f-g, f+g };
    return beat_freqs[0] *  beat_freqs[1] ;
}
```

6.2.18 Compound literals

Compound literals used to be called “Constructor Expressions” before ISO C99 normalized them under the term “Compound Literals”. A compound literal looks like a cast containing an initializer:

```
#include <stdio.h>
#include <malloc.h>

struct foo {int a; char b[2];} ;

struct foo * givefoo(int x, int y, char a, char b) {
  struct foo * sfoo = (struct foo *) malloc(sizeof (struct foo));
  /* Fill in the anonymous struct at once with a Compound Literal */
  *sfoo = (struct foo) {x + y, a, b};
  return sfoo;
}
```

GNU C allows initialization of objects with static storage duration by compound literals, whereas ISO C99 does not.

6.2.19 Designated initializers

This extension was called “GNU Style Labeled Elements in Initializers”. It is now an ISO C99 feature. It allows the initialization of particular elements of an aggregate, a structure or an array, by specifying the member name or the indices of the elements to initialize, in any order.

```
const int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
int a[6] = { [4] 29, [2] = 15 } ;
enum { v1 = 1, v2 = 2 , v4 = 4  } ;
int b[6] = { [1] = v1, v2, [4] = v4 } ;
struct point { int x, y; };
struct point  makep(int xvalue, int yvalue )
{
    struct point p = { y: yvalue, x: xvalue };
    return p ;
}
struct point  makepp(int xvalue, int yvalue )
{
    struct point p = { .y = yvalue, .x = xvalue };
    return p ;
}
```

With GNU C the = character can be omitted after the **[*index*]** indication.

6.2.20 Case ranges

Case ranges may be specified with integer value intervals in switch statements.

```
const char * which (int v) {
    switch (v) {
    case 0 ... 31: return "Control";
    case 'A' ... 'Z': return "Upper";
    case 'a' ... 'z': return "Lower";
    default:  return "None";
    }
}
```

6.2.21 Cast to a union type

A cast to union type is similar to other casts, except that the type specified is a union type. The type is specified either with the **union** tag or with a **typedef** name.

```
union foo { int i; double d; } u, v;
makefoo (int i, double f) {
  u = (union foo) i;
  v = (union foo) f;
}
```

6.2.22 Dollar signs in identifier names

Dollar signs are allowed in identifier names.

```
int $a;
```

6.2.23 Prototypes and old-style function definitions

GNU C extends ISO C to allow a function prototype to override a later old-style non-prototype definition.

```
int isroot (uid_t);

int isroot (x) /* ??? lossage here ??? */
    uid_t x;
{
    return x == 0;
}
```

6.2.24 C++ comments

```
// C++ comment
```

C++ comments are not recognized by the **st200cc** options **-ansi** or **-traditional**. This is to avoid problems with constructs that contain the forward slash character "**//**". For example:

```
x = a //**/b;
```

6.2.25 Character ESC in constants

The sequence "**\e**" is recognized in string or character constants as an ASCII <escape> character.

```
char escape = '\e';
char s[] = "\e\e";
```

6.2.26 Inquiring on alignment of types or variables

__alignof__ allows enquiries about how an object is aligned, or the minimum alignment required by a type or variable.

```
struct foo { int x; char y; } f;
int x = __alignof__ (double);
int b = __alignof__ (f.y);
```

6.2.27 Incomplete enum type

An **enum** type can be defined without specifying its possible values.

```
typedef enum _e e;

struct _s {
  e* p;
} s;

enum _e { red, green, blue, black };

e x;
```

6.2.28 Function names as strings

GNU **cc** predefines two magic identifiers to hold the name of the current function. The identifier **__FUNCTION__** holds the name of the function as it appears in the source. The identifier **__PRETTY_FUNCTION__** holds the name of the function pretty printed in a language specific fashion.

```
char here[] = "Function " __FUNCTION__ " in file " __FILE__;
```

6.3 Attributes

Generally, attributes are a much better design than a **#pragma** directive for a number of reasons. Firstly, an attribute specification is a piece of C language that can be generated by use of a **cpp** macro definition, whereas a **#pragma** directive generation is generally not supported by non-GNU C preprocessors. Secondly, it avoids the scoping issues of the **#pragma** directive.

Several attributes can be applied to the same object by using a comma to separate them. For example, to declare a symbol that is both weak and aliased:

```
void useful (void) __attribute__ ((weak, alias("useful_func")));
```

6.3.1 Placement and layout

section

Applied to function: Place a function in a user-defined section.

```
void myfunc (void) __attribute__ ((section(".mytext")));

void myfunc (void) {
  printf ("From myfunc in .mytext section.\n");
}
```

Applied to a data object: Place the data in a user-defined section.

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
```

Support must be explicitly added in the startup file or system loader to load the newly created section.

aligned

Applied to a variable or a structure field: Specifies a minimum alignment for a variable or structure field, measured in bytes. The **aligned** attribute can only increase the alignment; it can be decreased by specifying **packed** as well.

```
int x __attribute__ ((aligned (16))) = 0;
struct _s { int x[2] __attribute ((aligned (8))); };
short array [3] __attribute ((aligned));
```

Applied to a type:

```
typedef int more_aligned_int __attribute__ ((aligned(8)));
```

weak

Applied to a function: Causes the function to be emitted as a weak symbol. Set to **0** if the symbol is not defined at link time. This is primarily of use in defining library functions which can be overridden in user code:

```
void d_stub (void) __attribute__ ((weak));
if (d_stub) {
     d_stub();
}
```

Applied to data: Causes the declaration to be emitted as a **weak** symbol rather than a global symbol. This is primarily of use in defining variables which can be overridden in user code:

```
int debug  __attribute__ ((weak)) = 0;
```

alias

Applies only to functions: The required functionality is to provide an alias name for a given function. Often used in conjunction with the **weak** requirement to define an alternate **weak** name for a given function.

```
void useful_func (void) {
    /* ... Do something ... */
}

void useful (void) __attribute__ ((alias("useful_func")));
```

packed

Applies only to data: Specifies that a variable or structure should have the smallest possible alignment - one byte for a variable, and one bit for a field, unless a larger value with the **aligned** attribute is specified.

The specified data alignment is applied during data layout, and the code generator emits safe sequence of instructions to avoid causing a misalign trap.

```
struct foo { char a; int x __attribute__ ((packed)); };
```

used

The GCC manual specifies that the **used** attribute may only apply to functions. For **st200cc** it may also apply to variables:

- The **used** attribute, attached to a function, means that the code must be emitted for this function, even if this function appears never to be referenced.
- This attribute, attached to a variable, means that the definition must be emitted for the variable even if it appears that the variable is not referenced.

The **used** attribute follows the same syntax as any GCC attribute.

For a procedure:

```
static int Foo() __attribute__ ((used)) ;
```

For uninitialized data:

```
static foo  __attribute__((used)) ;
```

For initialized data:

```
static foo __attribute__((used)) = 2 ;
```

Note: The assembly has been specifically extended to support this attribute:

```
.type Foo, @function, used
.type foo, @object, used
```

A motivation for using this attribute is to avoid the deletion of an unreferenced symbol by the dead code, dead data or IPA optimization. This can be useful for debugging purposes (for instance a function dumping a specific data structure that is called only interactively from debugging sessions is removed if not marked '**used**', since the compiler does not find any reference to it).

constructor and destructor

Applies only to functions: The **constructor** attribute causes the function to be called automatically before execution enters **main()**. Similarly, the **destructor** attribute causes the function to be called automatically after **exit()**.

```
void initdata (void) __attribute__ ((constructor));
void terminatedata (void) __attribute__ ((destructor));
```

6.3.2 Optimization

This section applies only to functions.

noreturn

Enables a function to be declared that cannot return, such as **abort** or **exit**.

Useful indication to optimizers.

```
void byebye () __attribute__ ((noreturn));
```

malloc

Used to tell the compiler that a function returns a pointer that cannot alias anything. Useful indication to optimizers.

```
void * get_block (int) __attribute__ ((malloc));
```

6.3.3 Visibility attributes

The visibility attributes are supported as follows:

```
__attribute__((__visibility__("visibility-type")))
__attribute__((visibility("visibility-type")))
```

where ***visibility-type*** can be **"default"**, **"hidden"**, **"protected"**, **"internal"**

default

Default visibility is the normal case for ELF. This value is available for the visibility attribute to override other options that may change the assumed visibility of symbols.

hidden

Hidden visibility indicates that the symbol will not be placed into the dynamic symbol table, so that no other module (executable or shared library) can reference it directly.

protected

Protected visibility indicates that the symbol will be placed in the dynamic symbol table, but that references within the defining module will bind the local symbol. This means that the symbol cannot be overridden by another module.

internal

Internal visibility is like hidden visibility, but with additional processor-specific semantics. For the ST200, this means that the function is never called from another module.

Note: That hidden symbols, though they cannot be referenced directly by other modules can be referenced indirectly by function pointers. By indicating that a symbol cannot be called from outside the module, the compiler may for instance omit the load of a PIC register since it is known that the calling function has already defined the correct value.

6.3.4 Miscellaneous attributes

format

The **format** attribute specifies that a function takes **printf**, **scanf**, **strftime** or **strfmon** style arguments which should be type-checked against a format string.

```
extern int my_printf (void *my_object, const char *my_format, ...)
__attribute__ ((format(printf, 2, 3)));
```

format_arg

The **format_arg** attribute specifies that a function takes a format string for a **printf**, **scanf**, **strftime** or **strfmon** style function and modifies it, so that the result can be passed to a **printf**, **scanf**, **strftime** or **strfmon** style function.

```
extern char * my_dgettextprintf (void *my_domaint, const char
*my_format) __attribute__ ((format_arg(2)));
```

mode

This attribute specifies the data type for the declaration - whichever type corresponds to the mode. Refer to the *GNU Compiler Collection Internals* document for the definitions of modes, see http://gcc.gnu.org/onlinedocs/gccint.

Use the keywords **__byte__**, **__word__** and **__pointer__** to indicate the mode corresponding to these quantities.

```
unsigned int qi __attribute__ ((mode (QI)));
unsigned int w  __attribute__ ((mode (__word__)));
```

6.3.5 Built-ins

A built-in is used as a function call, but is expanded by the compiler very early in the intermediate representation, instead of doing a function call.

__builtin_constant_p

This built-in tests if a value is a constant at compile time.

```
int x;
#define C 1

int main () {
  if (__builtin_constant_p (C) == 1)
      printf ("c is proved to be a constant\n");
  if (__builtin_constant_p (x) == 0)
      printf ("x is a not proved to be a constant\n");
  return 0;
}
```

__builtin_return_address

__builtin_return_address gets the return address of the currently executing function.

```
void bar () {
   printf ("RA = 0x%08x\n", (int)__builtin_return_address (0));
}
```

__builtin_expect

```
long __builtin_expect (long exp, long c)
```

__builtin_expect provides the compiler with branch prediction information.

The return value is the value of **exp**, which should be an integral expression. The value of **c** must be a compile-time constant. The semantics of the built-in are that it is expected that **exp == c**.

For example:

```
if (__builtin_expect (exp, 0))
     foo ();
```

would indicate that we do not expect to call **foo**, since we expect **exp** to be zero.

__builtin_classify_type

__builtin_classify_type ignores the value of the object, considering only its data type. It returns an enum describing what kind of type ***object*** is.

```
enum type_class __builtin_classify_type(object)

enum type_class
{
   no_type_class = -1,
   void_type_class, integer_type_class, char_type_class,
   enumeral_type_class, boolean_type_class,
   pointer_type_class, reference_type_class, offset_type_class,
   real_type_class, complex_type_class,
   function_type_class, method_type_class,
   record_type_class, union_type_class,
   array_type_class, string_type_class, set_type_class,
   file_type_class, lang_type_class
};
```

__builtin_prefetch

```
void __builtin_prefetch(const void *addr, ...)
```

This function causes the compiler to generate a **pft** instruction for address ***addr***. There are two optional arguments: ***rw*** and ***locality***. These arguments are currently ignored by the ST200 compiler.

Automatic prefetching is not performed upon arrays that have the built-in function **__builtin_prefetch** applied to them.

7 GNU ASM

7.1 Introduction

st200cc compiler accepts “extended inline assembly” **asm**, as part of C programs. This chapter only summarizes the main features of the **asm** implementation and describes its limitations, it is not a substitute for the GNU documentation.

7.1.1 Syntax

```
asm (template : output operands : input operands : clobber list) ;
```

or

```
__asm__ (template : output operands : input operands : clobber list) ;
```

- **template**: the assembler instruction, defined as a string constant,
- **output operands**: a list of comma separated output operands,
- **input operands**: a list of comma separated input operands,
- **clobber list**: a list of comma separated clobbered operands.

The template section contains plain assembler, and uses ordinary ST200 assembler syntax, with the notable exception of the **%*i*** (***i*** is a positive integer) notation that refers to the i^{th} output operand or input operand.

Note: Multiple consecutive strings are automatically concatenated and enable a readable and correct template input. Multiple assembler instructions can be put together in a single **asm** *template, separated by explicit newline characters ‘***\n***’.*

If there are no output operands but there are input operands, two consecutive colons **must** be placed where the output operands would go.

In the output and input list:

- each operand is described by an “operand constraint string” followed by a C expression in parentheses,
- the available constraints are the following:
  - **r** general purpose register operand,
  - **b** boolean register operand,
  - **i** immediate integer operand, including symbolic constants only known at assembly time,
  - **n** immediate integer operand, known at compile time,
  - **g** general purpose register or immediate integer,
- an operand constraint can be prefixed by the following modifiers:
  - **=** write-only operand, used for output operands,
  - **&** early clobber operand, does not prevent the use of **=**.

In the clobber list:

- general registers are referred to by ***ri*** (where ***i*** has the range [0,63]), they map to the corresponding R***i*** hardware registers,
- branch registers are referred to by **b*i*** (where ***i*** has the range [0,7]), they map to the corresponding B***i*** hardware boolean registers.

7.1.2 Assumptions

- Output operand expressions must be **lvalue**s.
- The compiler assumes that the input is consumed before the outputs are produced, unless an output operand has the ‘**&**’ constraint modifier (also called “early clobber”). The compiler will not assign the same register to an input operand and an early-clobber operand. However, the compiler may assign the same register to an input operand and to a non-early-clobber output operand.

7.1.3 Volatile

The volatile syntax is as follows:

```
asm volatile (template : output operands : input operands : clobber
list);
```

or

```
__asm__ volatile (template : output operands : input operands : clobber
list);
```

The **volatile** keyword indicates that an instruction has side effects. A **volatile** statement will not be deleted if it is reachable. The order of **volatile asm** statements and, or other **volatile** accesses will be preserved. A consecutive sequence of **volatile asm** statements may not stay perfectly consecutive, since some other instructions may be scheduled in between.To achieve the effect of keeping instructions perfectly consecutive, use a single **asm** instruction.

An **asm** statement without any operand or clobbers will be treated identically to a **volatile asm** statement, the same as for an **asm** statement without an output operand.

7.1.4 Scheduling considerations

On the ST220 processor, scheduling considerations must account for the absence of hardware interlocks. The programmer must make sure that all **asm** outputs are available in the next cycle following the **asm**, for instance by adding explicit (possibly empty) bundles.

On the ST231 processor, this is not necessary as this processor is interlocked.

Generating different schedules for these different processors can be handled by using the core symbol defined by the C preprocessor (**__st220__**, **__st231__**).

In any case, correct instruction bundling must occur (for example, bundling constraints and multiplication alignments).

7.1.5 Restrictions

- The compiler does not parse the assembler instruction template, this means that it does not check if it is valid assembler input.
- Up to 10 operands, results and clobbered registers are allowed.
- Multiple alternative constraints are not supported.
- At **-O3** optimization level, the loop nest optimizer is disabled for loops containing **asm** statements.

7.1.6 Example

The following function adds two unsigned integers, completes a carry and obtains the 33-bit result through the **addcg** ST200 instruction. The output is represented by 64 bits. It uses both ordinary registers and boolean registers as input and output, and shows that **struct** members can be used as C expressions to receive the computed result.

```
inline unsigned long long addcarry(unsigned int t, unsigned int s,
unsigned int c) {
   typedef union {
      unsigned long long v_ ;
#if defined(__LITTLE_ENDIAN__)
      struct { unsigned int l_ ; unsigned int h_ ; } lh ;
#elif defined(__BIG_ENDIAN__)
      struct { unsigned int h_ ; unsigned int l_ ; } lh ;
#else
#error "addcarry : Unknown endianness"
#endif
   } __ui64 ;
   __ui64 ui64v_ ;

   asm("addcg %0, %1 = %2, %3, %4"
      : "=r" (ui64v_.lh.l_) , "=b" (ui64v_.lh.h_)
      : "r" (t), "r" (s), "b" (c)
      ) ;
   return  ui64v_.v_ ;
}
```

8 Intrinsic functions

8.1 Introduction

8.1.1 Rationale

The **st200cc** compiler recognizes a number of intrinsic operators which can be used to produce assembly language statements that otherwise could not be expressed through standard ANSI C/C++.

These intrinsics are specified and called just like standard ANSI C/C++ functions using standard types, however, they are treated specially by the compiler. The ST200 intrinsics apply to the ST220 and ST231 parts.

8.1.2 Models

ST200 intrinsics have been modelled as C functions, which act as executable specifications. This has the benefit that models can be used to develop DSP algorithms on a workstation, that are immediately and safely ported to the ST220 and ST231.

The implementation of the models **`imodels.c`** is delivered with the current compiler distribution.

8.2 Naming intrinsics

8.2.1 Naming scheme

STMicroelectronics intrinsics may be used for several families of DSP and micro controllers whose natural integral types are not always equivalent.

To accommodate a variety of targets, a naming scheme has been designed which takes into account this variety. An intrinsic has basically a generic name containing its signature:

__{return_type}_{operation}_{arguments_types}+

Intrinsics are given shorter and more mnemonic alias names for each target, taking into account the natural names of types on the specific target.

For instance, the natural name of the 32-bit integer on the ST200 is **word**, so the intrinsic that realizes the clamped addition of two 32 bit integers is named **__int32_addc_int32_int32** in its generic form and **__addcw** in its ST200 form. Where the **addc** part denotes the clamped addition operator, and **w** the fact that the result is of word type.

Note: The presence of the two leading underscores on each name denotes (according to the ISO/IEC 9899 C Standard) that no such name should be defined by the user.

More specifically:

"All identifiers that begin with an underscore and either an upper case letter or another underscore are always reserved for any use."

Finally the International Telecommunication Union (ITU) and European Telecommunications Standard Institute (ETSI)[1] basic operators have their own naming scheme, thus they must be explicitly mapped onto ST200 intrinsics. This is explained further in *Operations on page 127*.

8.2.2 Building intrinsic names

This section elaborates on the rules for building intrinsic names. It is important to understand the principles behind the naming conventions, because of the number of intrinsic functions that are available. Understanding these principles relieves the need to remember specific names, and enables the name of an intrinsic to be deduced from its specific operation.

The following sections expand on the type mapping and name conventions used in the intrinsics.

1. Many of the standard algorithms in Wireless and Wireline communications such as GSM (Global System for Mobile Telecommunications) or AMR-NB for UMTS (Adaptive Multi Rate Narrow Band for UMTS) encoders, are provided by the ITU and ETSI, as ANSI C code using 16-bits fractional arithmetic. For the ITU refer to http://www.itu.int or for the ETSI, refer to http://www.etsi.org.

   To specify the fractional arithmetic model which is foreign to ANSI C, a set of subroutines are used implementing basic fractional operations. As these basic fractional operations are provided as ANSI C source codes, an algorithm complying with the ITU or ETSI compiles on any ANSI C compiler.

   It therefore requires little effort to compile and run the algorithm on any workstation or DSP. However, an efficient implementation on a DSP requires some modifications of the original code. The **st200cc** compiler provides a set of intrinsics that help make these modifications as minor as possible.

Types

Table 31 summarizes the type names use in the intrinsics description.

Description	Name	Models mapping	ST200 mapping (size)	ST200 notation
-	**int16**	short	short (16)	**h**
-	**uint16**	unsigned short	unsigned short (16)	**uh**
Fractional q15	**fract16**	short	short (16)	**h** (equivalent to signed short)
-	**int32**	int	int (32)	**w**
-	**uint32**	unsigned int	unsigned int (32)	**uw**
Fractional q31	**fract32**	int	int (32)	**w** (equivalent to signed int)
Fractional q63	**fract64**	long long	long long (64)	**l**
-	**int64**	long long	long long (64)	**l**
-	**uint64**	unsigned long long	unsigned long long (64)	**ul**

Table 31: ST220 intrinsic type names

Note: 1 These types are defined in the ***st220types.h*** *files, which is part of the compiler distribution. This file applies to both the ST220 and ST231.*

2 Operations on fractions are specifically denoted by the operator name with the addition of the ***f*** *suffix to the standard name.*

3 Using models requires a full 64-bit implementation of the ***long long*** *and* ***unsigned long long*** *types.*

The model requires the standard C99 **long long** type. The standard PC compiler uses **__int64** to support a full 64-bit type. Other compilers can be obtained that do support the **long long** type, for example a **gcc** based compiler. Refer to http://sources.redhat.com/cygwin.

Operations

Table 32 summarizes the operator names used in the intrinsics description.

The reader must be familiar with the concepts of modular, saturating, rounding, and fractional arithmetic. Pay particular attention to the Remarks column in *Table 32*.

Model stem	Description	Remarks
add	Modular addition	
addc	Saturating addition	
sub	Modular subtraction	
subc	Saturating subtraction	
neg	Modular negate	
negc	Saturating negate	
min	Minimum	
max	Maximum	
abs	Modular absolute value	
absc	Saturating absolute value	
dist	Distance	
shr	Right shift	Positive shift amount, shifting more than integral operand size in bits is implementation defined
shrr	Rounded right shift	
shlc	Saturating left shift	Positive shift amount, shifting more than integral operand size in bits is implementation defined
mul	Modular multiplication	
mulc	Saturating multiplication	
mulrc	Rounded saturating multiplication	

Table 32: ST200 intrinsic operator names

Model stem	Description	Remarks
mulh	High multiplication	High part of classical multiplication
div	Division	Sign of Quotient is sign of algebraic quotient, if inexact |q| < |D/d|
divc	Saturating fractional division	
mod	Modulus	When Quotient q is defined, q * d + Modulus = D
modc	Saturating fractional modulus	
bitcnt	Population count	
bitrev	Bit reverse	
rotl	Rotate left	
xshl	Cross-shift left	
xshr	Cross-shift right	
edges	Edges	
lzcnt	Leading zero count	
prior	Get left shifts to normalize	
norm	ETSI get left shifts to normalize	
clamp	Saturation	
round	Rounding	
get	Get high/low part	
put	Put in high/low part	
bitclr	Bit clear	
bitset	Bit set	
bitnot	Bit complement	
bitval	Bit test	

Table 32: ST200 intrinsic operator names

Model stem	Description	Remarks
inseq	Compare for equal and insert	
insne	Compare for not equal and insert	
insgt	Compare for greater and insert	
insge	Compare for greater or equal and insert	
inlts	Compare for less and insert	
insle	Compare for less or equal and insert	

Table 32: ST200 intrinsic operator names

For example, searching for an operator that computes the maximum value of two 32-bit integers is quite simple.

1 Operator stem is **max**, and there is no specific ST200 name.

2 Operands are both of **int32** type, so the suffix is **int32_int32**.

3 Result type is **int32**, so the prefix is **int32**.

4 Therefore, the generic model name of the operator is **__int32_max_int32_int32**.

To get the specific ST200 operator name is also quite simple.

1 Operator stem is **max** (there is no specific ST200 name).

2 Operand and result type are identical, **int32** type, therefore the suffix **w** is used.

3 Therefore, the generic model name of the operator is **__maxw**.

Note: All operators have been quite thoroughly defined, to avoid redundancies, and are limited to useful combinations. There is no need, for example, to have a fractional addition operator, because addition of fractional types using the same representation is equivalent to addition on ordinary integral types.

8.3 Using intrinsics from C/C++

8.3.1 Include files

This section explains the usage of include files that are peculiar to intrinsics.

Note: All techniques described here apply to both C and C++, and the headers are both C and C++ compatible.

Using intrinsics on an ST200 platform

ST200 intrinsics prototypes are available in the **st220.h** include file.

Note: All header files mentioned in this chapter apply to both the ST220 and ST231.

To include intrinsics in an application use:

```
#include <st220.h>
```

If ETSI names are required use the following as well:

```
#include <etsitost220.h>
```

This injects a set of macros that renames original ETSI/ITU basic operators.

Most of these macros are just name defines, with the exception of shift operators that are macros with parameters. This means that it is not possible to take the address of shift operators.

Note: Including **st220.h** *implicitly includes* **st220types.h** *which defines the appropriate basic types.*

Using intrinsics models

Prototypes for the intrinsic models are provided in **imodels.h** and their implementation is in **imodels.c**

To include an intrinsics model in an application use:

```
#include <imodels.h>
```

If ETSI names are required use the following as well:

```
#include <etsitom.h>
```

Note: Including **imodels.h** *implicitly includes* **st220types.h** *which defines the appropriate basic types.*

Care must also be taken to link with the **imodels.o** file, produced on the specific platform.

Switching between ST200 intrinsics and modelled intrinsics

Is is often extremely useful to be able to keep the same source baseline, either running with models on a workstation, or on an ST200 platform.

For example, on an ST220 this is done using the implicitly defined **__st220__** symbol:

```
#ifdef __st220__
#include <st220.h>
#include <etsitost220.h>
#else    /* !__st220__ */
#include <imodels.h>
#include <etsitom.h>
#endif   /* __st220__ */
```

Note: 1 It is possible to remap names in both directions.

2 For an ST231 platform, the symbol **__st231__** *would be used with the same ST220 header files, as these apply equally to ST231.*

An application written exclusively using the model names can be retargeted immediately to ST200 by including:

```
#include <mtost220.h>
```

This has the effect of renaming all models to ST200 intrinsics names.

An application written exclusively using the ST200 names can easily be switched to using model names by including:

```
#include <st220tom.h>
```

This has the effect of renaming all ST200 intrinsic names to model names.

To allow the remapping of names, use one set of names exclusively, that is use either the model names or the ST200 intrinsic names.

Example

The following example demonstrates the usage of ST200 intrinsics in the application source code while maintaining the ability to compile on a generic target by switching automatically back to models.

```
#if defined(__st200__)
/* Include intrinsics definitions when target is ST200 */
#include <st220.h>
#else /* !__st200__ */
/* Include models and renaming scheme when target is unknown */
#include <imodels.h>
#include <st220tom.h>
#endif /* __st200__ */

unsigned int SwapBytes(unsigned int x)
{
   /*
    * On ST200, this will be emitted as a single instruction
    * bswap $r16 = $r16
    *
    * On other targets, thanks to <st220tom.h> macros,
    * this name will be replaced by the generic name
    * and looked up in the models source files.
    * Thus the generated code will be a call to
    *__uint32_swapb_uint32
    * To compile on a workstation with a generic compiler
    * (for instance gcc on Solaris or Linux) you will have to
    *  include <tools-dir>/host/imodels
    *  in your include search path
    * (use the -I compiler option )
    *  to add <tools-dir>/host/imodels/imodels.c to your source
    * files */
   return __swapbw(x) ;
}
```

When compiled with the **st200cc** compiler, the following code provides maximum efficiency for the ST200 target, it can be compiled for any other target and automatically guarantees the same execution semantics.

With the **st200cc** compiler, an application using this file **swapb.c** can be compiled without modification (apart from defining the toolset installation path).

```
$ st200cc swapb.c other-files.c -o main.u
```

With a gnu **gcc** compiler on Unix:

```
$ gcc -I<tools-dir>/include -I<tools-dir>/host/imodels <tools-dir>/
host/imodels/imodels.c swapb.c other-files.c -o main.sol
```

Note: the installation path of the ST200 toolset must be defined, this example assumes that the **$PATH** environment variable has been updated with **<*tools-dir*>**.

Finally, if this example is to be used in real code, it would be better to define the **SwapBytes** function as inline, and provide it through an appropriate header file.

8.4 Understanding intrinsic models

The inner workings of an intrinsic can be difficult to understand. In order to understand the precise operating semantics it helps to understand the model implementation that intrinsics are based on.

The following sections describe the fundamental operators that the models are based on, then gives an implementation example.

8.4.1 Understanding fundamentals operators

The following macro operators are defined:

- **EXT(*x*, *n*)** normalizes ***x*** considered signed on ***n*** bits in a **long long**,
- **EXTU(*x*, *n*)** normalizes ***x*** considered unsigned on ***n*** bits in a **long long**,
- **CLAMP(*x*, *n*)** saturates ***x*** on ***n*** bits in a **long long**.

All models are based on a systematic and consistent use of the operators which enables understanding of the inner workings of intrinsics.

8.4.2 Understanding models

Consider the example of the **__int16_addc_int16_int16** operator:

```
int16 __int16_addc_int16_int16(int16 r0, int16 r1) {
    long long t0 = EXT(r0, 16);
    long long t1 = EXT(r1, 16);
    long long t2 = t0 + t1;
    long long t3 = CLAMP(t2, 16);
    return t3;
}
```

In this example, inputs are normalized to **long long**, through the **EXT** operations, keeping 16 bits signed precision. Then the addition is computed with full precision and finally the result is saturated to 16 bits by the **CLAMP** operator.

8.5 Intrinsic functions summary

The tables shown in *Section 8.5.1* provide the names of all ST200 intrinsics functions along with their model names and properties.

The tables contain:

- the model name of the intrinsic,
- the purpose of the intrinsic:
  - C operator: supports C arithmetic,
  - C runtime arithmetic helper: supports optimized C arithmetic,
  - DSP operator: saturating/fractional arithmetic,
  - ETSI/ITU operator: standard conformance operator,
- the ST200 specific name.

The intrinsic functions are generally categorized by operator. Within each category they are grouped by **operand** and **result** type. All the compiler-generated arithmetic intrinsics are held in the library **libgcc.a**.

8.5.1 Functions

Intrinsic	Purpose	ST200 name
`__int32_addc_int32_int32`	ETSI L_add	`__addcw`
`__int16_addc_int16_int16`	ETSI add	`__addch`

Table 33: Add intrinsics

Intrinsic	Purpose	ST200 name
`__int32_subc_int32_int32`	ETSI L_sub	`__subcw`
`__int16_subc_int16_int16`	ETSI sub	`__subch`

Table 34: Subtract intrinsics

Intrinsic	Purpose	ST200 name
`__int32_negc_int32`	ETSI L_negate	`__negcw`
`__int16_negc_int16`	ETSI negate	`__negch`

Table 35: Negate intrinsics

Intrinsic	Purpose	ST200 name
`__int32_min_int32_int32`	C optimization	`__minw`
`__int16_min_int16_int16`	C optimization	`__minh`
`__uint32_min_uint32_uint32`	C optimization	`__minuw`
`__uint16_min_uint16_uint16`	C optimization	`__minuh`

Table 36: Minimum intrinsics

Intrinsic	Purpose	ST200 name
`__int32_max_int32_int32`	C optimization	`__maxw`
`__int16_max_int16_int16`	C optimization	`__maxh`
`__uint32_max_uint32_uint32`	C optimization	`__maxuw`
`__uint16_max_uint16_uint16`	C optimization	`__maxuh`

Table 37: Maximum intrinsics

Intrinsic	Purpose	ST200 name
`__int32_abs_int32`	C optimization	`__absw`
`__int16_abs_int16`	C optimization	`__absh`
`__int32_absc_int32`	ETSI L_abs	`__abscw`
`__int16_absc_int16`	ETSI abs_s	`__absch`

Table 38: Absolute value intrinsics

Intrinsic	Purpose	ST200 name
`__int32_dist_int32_int32`	DSP operator	`__distw`
`__int16_dist_int16_int16`	DSP operator	`__disth`
`__uint32_dist_uint32_uint32`	DSP operator	`__distuw`
`__uint16_dist_uint16_uint16`	DSP operator	`__distuh`

Table 39: Absolute distance intrinsics

Intrinsic	Purpose[a]	ST200 name
`__int32_shrr_int32_uint16`	ETSI L_shr_r	`__shrrw`
`__int16_shrr_int16_uint16`	ETSI shr_r	`__shrrh`

Table 40: Right shift intrinsics

a. To achieve the ETSI operator, a combination of the intrinsic functions are required. For example, L_shl in *Table 41* is implemented in terms of `__shlcw` and `__shrw`.

Intrinsic	Purpose[a]	ST200 name
`__int32_shlc_int32_uint16`	ETSI L_shl	`__shlcw`
`__int16_shlc_int16_uint16`	ETSI shl	`__shlch`

Table 41: Left shift intrinsics

Intrinsic	Purpose	ST200 name
`__int64_mul_int32_int32`	C optimization	`__muln`
`__uint64_mul_uint32_uint32`	C optimization	`__mulun`
`__int32_mul_int32_int16`	C optimization	`__mulm`
`__int16_mul_int16_int16`	C optimization	`__mulh`
`__uint32_mul_uint32_uint16`	C optimization	`__mulum`
`__uint16_mul_uint16_uint16`	C optimization	`__muluh`
`__fract32_mulc_fract32_fract32`	DSP operator	`__mulfcw`
`__fract32_mulc_fract32_fract16`	DSP operator	`__mulfcm`
`__fract16_mulc_fract16_fract16`	ETSI mult	`__mulfch`

Table 42: Multiply intrinsics

Intrinsic	Purpose	ST200 name
`__int32_div_int32_int16`	C optimization	`__divm`
`__int16_div_int16_int16`	C optimization	`__divh`
`__uint32_div_uint32_uint16`	C optimization	`__divum`
`__uint16_div_uint16_uint16`	C optimization	`__divuh`
`__fract32_divc_fract32_fract32`	DSP operator	`__divfcw`
`__fract32_divc_fract32_fract16`	DSP operator	`__divfcm`
`__fract16_divc_fract16_fract16`	ETSI div_s	`__divfch`

Table 43: Divide intrinsics

Intrinsic	Purpose	ST200 name
`__int32_mod_int32_int16`	C optimization	`__modm`
`__int16_mod_int16_int16`	C optimization	`__modh`
`__uint32_mod_uint32_uint16`	C optimization	`__modum`
`__uint16_mod_uint16_uint16`	C optimization	`__moduh`
`__fract32_modc_fract32_fract32`	DSP operator	`__modfcw`
`__fract32_modc_fract32_fract16`	DSP operator	`__modfcm`
`__fract16_modc_fract16_fract16`	ETSI mod_s	`__modfch`

Table 44: Modulus intrinsics

Intrinsic	Purpose	ST200 name
`__int64_mul_int32_int16`	C optimization	`__mpml`
`__uint64_mul_uint32_uint16`	C optimization	`__mpuml`
`__fract32_mulc_fract16_fract16`	ETSI L_mult	`__mpfcw`
`__fract16_mulrc_fract16_fract16`	ETSI mult_r	`__mpfrch`

Table 45: Multiply special intrinsics

Intrinsic	Purpose	ST200 name
`__int32_mulh_int32_int32`	C Runtime Arithmetic helper	`__mulhw`
`__int16_mulh_int16_int16`	C Runtime Arithmetic helper	`__mulhh`
`__uint32_mulh_uint32_uint32`	C Runtime Arithmetic helper	`__mulhuw`
`__uint16_mulh_uint16_uint16`	C Runtime Arithmetic helper	`__mulhuh`

Table 46: Multiply high intrinsics

Intrinsic	Purpose	ST200 name
`__uint32_edges_uint32_uint32`	DSP operator	`__edgesw`
`__uint16_edges_uint16_uint16`	DSP operator	`__edgesh`
`__uint32_rotl_uint32_uint16`	DSP operator	`__rotlw`
`__uint16_rotl_uint16_uint16`	DSP operator	`__rotlh`
`__uint32_xshl_uint32_uint32_uint16`	DSP operator	`__xshlw`
`__uint16_xshl_uint16_uint16_uint16`	DSP operator	`__xshlh`
`__uint32_xshr_uint32_uint32_uint16`	DSP operator	`__xshrw`
`__uint16_xshr_uint16_uint16_uint16`	DSP operator	`__xshrh`
`__uint16_xshr_uint16_uint16_uint16`	DSP operator	`__priorl`
`__int16_prior_int32`	DSP operator	`__priorw`
`__int16_prior_int16`	DSP operator	`__priorh`
`__int16_norm_int64`	ETSI norm_l	`__norml`
`__int16_norm_int32`	ETSI norm_l	`__normw`

Table 47: Miscellaneous intrinsics

Intrinsic	Purpose	ST200 name
`__int16_norm_int16`	ETSI norm_s	`__normh`
`__uint16_lzcnt_int64`	DSP operator	`__lzcntl`
`__uint16_lzcnt_int32`	DSP operator	`__lzcntw`
`__uint16_lzcnt_int16`	DSP operator	`__lzcnth`

Table 47: Miscellaneous intrinsics

Intrinsic	Purpose	ST200 name
`__int16_clamp_int32`	ETSI saturate	`__clampwh`
`__int16_near_int32`	IEEE round	`__nearwh`
`__int16_nearc_int32`	IEEE round and clamp	`__nearcwh`
`__int16_round_int32`	DSP operator	`__roundwh`
`__int16_roundc_int32`	ETSI round	`__roundcwh`
`__int32_puth_int16`	DSP operator	`__puthw`
`__int32_putl_int16`	DSP operator	`__putlw`
`__int16_geth_int32`	ETSI extract_h	`__gethh`
`__int16_getl_int32`	ETSI extract_l	`__getlh`

Table 48: Clamp, round, insert and extract intrinsics

Intrinsic	Purpose	ST200 name
`__uint32_bitclr_uint32_uint16`	DSP operator	`__bitclrw`
`__uint16_bitclr_uint16_uint16`	DSP operator	`__bitclrh`
`__uint32_bitset_uint32_uint16`	DSP operator	`__bitsetw`
`__uint16_bitset_uint16_uint16`	DSP operator	`__bitseth`
`__uint32_bitnot_uint32_uint16`	DSP operator	`__bitnotw`

Table 49: Bit manipulation intrinsics

Intrinsic	Purpose	ST200 name
`__uint16_bitnot_uint16_uint16`	DSP operator	`__bitnoth`
`__uint16_bitval_uint32_uint16`	DSP operator	`__bitvalw`
`__uint16_bitval_uint16_uint16`	DSP operator	`__bitvalh`
`__int32_bitrev_uint16_int32`	DSP operator	`__bitrevw`

Table 49: Bit manipulation intrinsics

Intrinsic	Purpose	ST200 name
`__int32_inseq_int32_int32_int32`	DSP operator	`__inseqw`
`__int32_insne_int32_int32_int32`	DSP operator	`__insnew`
`__int32_insgt_int32_int32_int32`	DSP operator	`__insgtw`
`__int32_insge_int32_int32_int32`	DSP operator	`__insgew`
`__int32_inslt_int32_int32_int32`	DSP operator	`__insltw`
`__int32_insle_int32_int32_int32`	DSP operator	`__inslew`
`__uint32_inseq_uint32_uint32_uint32`	DSP operator	`__insequw`
`__uint32_insne_uint32_uint32_uint32`	DSP operator	`__insneuw`
`__uint32_insgt_uint32_uint32_uint32`	DSP operator	`__insgtuw`
`__uint32_insge_uint32_uint32_uint32`	DSP operator	`__insgeuw`
`__uint32_inslt_uint32_uint32_uint32`	DSP operator	`__insltuw`
`__uint32_insle_uint32_uint32_uint32`	DSP operator	`__insleuw`

Table 50: Viterbi intrinsics

Intrinsic	Purpose	ST200 name
`__int32_mul_int32_int32`	C operator	`__mulw`
`__uint32_mul_uint32_uint32`	C operator	`__muluw`
`__int32_div_int32_int32`	C operator	`__divw`
`__uint32_div_uint32_uint32`	C operator	`__divuw`
`__int32_mod_int32_int32`	C operator	`__modw`
`__uint32_mod_uint32_uint32`	C operator	`__moduw`

Table 51: Integer 32 bit support

Intrinsic	Purpose	ST200 name
`__int64_add_int64_int64`	C operator	`__addl`
`__uint64_add_uint64_uint64`	C operator	`__addul`
`__int64_sub_int64_int64`	C operator	`__subl`
`__uint64_sub_uint64_uint64`	C operator	`__subul`
`__int64_neg_int64`	C operator	`__negl`
`__uint64_neg_uint64`	C operator	`__negul`
`__int64_shr_int64_uint16`	C operator	`__shrl`
`__uint64_shr_uint64_uint16`	C operator	`__shrul`
`__int64_shl_int64_uint16`	C operator	`__shll`
`__uint64_shl_uint64_uint16`	C operator	`__shlul`
`__int64_mul_int64_int64`	C operator	`__mull`
`__uint64_mul_uint64_uint64`	C operator	`__mulul`
`__int64_div_int64_int64`	C operator	`__divl`
`__uint64_div_uint64_uint64`	C operator	`__divul`
`__int64_mod_int64_int64`	C operator	`__modl`
`__uint64_mod_uint64_uint64`	C operator	`__modul`

Table 52: Long long support

Intrinsic	Purpose	ST200 name
`__int64_addc_int64_int64`	DSP operator [D_add]	`__addcl`
`__int64_subc_int64_int64`	DSP operator [D_sub]	`__subcl`
`__int64_negc_int64`	DSP operator [D_neg]	`__negcl`
`__int64_absc_int64`	DSP operator [D_abs]	`__abscl`
`__int64_max_int64_int64`	DSP operator	`__maxl`
`__uint64_max_uint64_uint64`	DSP operator	`__maxul`
`__int64_min_int64_int64`	DSP operator	`__minl`
`__uint64_min_uint64_uint64`	DSP operator	`__minul`
`__int32_clamp_int64`	DSP operator [D_sat]	`__clamplw`
`__int32_round_int64`	DSP operator [D_round]	`__roundlw`
`__int32_roundc_int64`	DSP operator [D_round]	`__roundclw`
`__int32_near_int64`	IEEE operator	`__nearlw`
`__int32_nearc_int64`	IEEE operator	`__nearclw`
`__fract64_mulc_fract32_fract32`	DSP operator [D_mult]	`__mpfcwl`
`__fract64_mul_fract32_fract16`	DSP operator	`__mpfml`
`__int64_puth_int32`	DSP operator	`__puthl`
`__int64_putl_int32`	DSP operator	`__putll`
`__int32_geth_int64`	DSP operator [D_extract_l]	`__gethw`
`__int32_getl_int64`	DSP operator [D_extract_h]	`__getlw`

Table 53: High precision fractional arithmetic

Intrinsic	Purpose	ST200 Name
`__uint16_swapb_uint16`	DSP operator	`__swapbh`
`__uint32_swaph_uint32`	DSP operator	`__swaphw`
`__uint32_swapb_uint32`	DSP operator	`__swapbw`

Table 54: Bit and byte ordering

Intrinsic	Purpose	ST200 Name
`__uint16_bitcnt_uint32`	DSP operator	`__bitcntw`
`__uint16_bitcnt_uint16`	DSP operator	`__bitcnth`

Table 55: Population count

8.6 ST231 intrinsics

This compiler supports the following new intrinsics:

```
int __st200mul32(int, int);
int __st200mul64h(int, int);
unsigned int __st200mul64hu(unsigned int, unsigned int);
int __st200mulfrac(int, int);
```

These intrinsics were originally developed to give quick access to new ST231 instructions, when they were not selected by the compiler. However, the three integer multiplication intrinsics are now almost useless since the compiler correctly selects the new instruction. The last intrinsic (**__st200mulfrac**) is never selected by the compiler, since its semantic is complex (multiply fractional with saturation and rounding).

To simplify development work, these intrinsics are available for any ST200 core.

They are expanded:

- as a single machine instruction when the target is **st231** (for example, when **-mcore=st231** command-line is used),
- as a sequence of instructions that emulate the behavior of the **st231** native instructions for other cores (**st220**).

Table 56 summarizes these intrinsics properties either when emulated or native:

Name	Instructions / critical path for emulation (ST220)	Instructions / critical path for ST231
__st200mul32	3 / 2	1 / 1
__st200mul64h	7 / 4	1 / 1
__st200mul64hu	13 / 6	1 / 1
__st200mulfrac	17 / 8	1 / 1

Table 56: ST231 intrinsics properties

8.7 Division and modulus built-ins

The compiler supports several division and modulus built-ins that expand to a loopless sequence of instructions to compute the result of the integer division (or modulus) operands for signed or unsigned integers.

These built-ins only exist for the integer division (or modulus) operator:

```
__builtin__divw
__builtin__divuw
__builtin__modw
__builtin__moduw
```

The benefits are that the division operators do not suffer from instruction call penalties (and possible related ICache conflicts), and that other instructions can be scheduled at the same time to maximize resource usage.

The drawback is that contrary to the library version, these operators do not minimize the number of iterations, whereas the library version considers the respective magnitude of the operands to minimize the division cycles when beneficial.

Note: These division and modulus operators should not be used when the divisor is a constant since the compiler emits efficient inline code when the constant is a power of 2.

Table 57 summarizes the properties of these built-ins.

Name	Instructions / critical path
`__builtin__divw`	`88 / 41`
`__builtin__divuw`	`76 / 39`
`__builtin__modw`	`77 / 40`
`__builtin__moduw`	`78 / 39`

Table 57: ST231 intrinsics properties

These built-ins incur a high cost in code size, and should probably be inlined only in critical inner loops.

9 Compiler bugs

9.1 Introduction

This chapter describes the different categories of compiler bugs and how they should be reported to STMicroelectronics.

9.2 Identifying a compiler bug

9.2.1 Category 1

The following cases are compiler or toolset bugs:

- the compilation phase ends with an assertion message,
- the compilation phase ends with a system error message (core dump, bus error),
- the compilation phase produces an output that cannot be assembled,
- the compilation phase never ends, or at least does not end in a reasonable amount of time,
- the compiler produces an error message for code that is valid input,
- the compiler produces code that does not compute the expected results (but see *Section 9.2.2*).

9.2.2 Category 2

The following case is possibly not a compiler or toolset bug.

The code is functional under a specific optimization level, but not under another. This may be due to an existing code bug that is only exposed by aggressive optimization.

9.3 Checks performed by user

The following checks should be performed on your code before reporting a bug:

- check that the code works correctly on at least one other compiler, on another host,
- check that the code does not access out-of-bound memory,
- check that the source code does not raise any warning when compiled with the **`-Wall`** option,
- check that the source code does not make assumptions that may be false: specifically check restrict annotations, and optimization pragmas,
- check that the code does not exercise language edges or does not violate language standards: an example of undefined behavior is to assume a specific behavior of shift operators when the shift amount is negative or bigger than the size of the type shifted.

9.4 Work-around

The following can be carried out to temporarily work-around a compiler bug.

- Demote the optimization level to **`-O1`** or **`-O0`** when compiling the specific file creating the problem, either in category 1 or 2. (See *Section 9.2.1* and *Section 9.2.2*.)
- Remove the optimization pragmas or restrict annotations.
- Finally, check that you have an up-to-date compiler release.

9.5 Reporting a compiler bug

Carry out the following if a compiler bug is encountered.

- Obtain your compiler version by running the command **`st200cc -version`**.
- If the compiler bug is in category 1 (see *Section 9.2.1 on page 147*), prepare a pre-processed input file that can reproduce the problem.
- If the compiler bug is in category 2 (see *Section 9.2.2 on page 148*), prepare a source set and Makefile that can reproduce the problem.
- Supply the full command line that generates the problem.
- Report the result of the following command in the shell that you use: **`uname -a`**.
- Prepare a description of the expected result and the actual result.
- Report all the above information through your local ST Field Applications Engineer (FAE).

Finally, when in doubt, it is preferable that a possible bug is reported than forgotten.

9.6 Known bugs and limitations

Please refer to the *Bug list* supplied with the toolset on the CD for an up-to-date list of bugs and limitations.

10 ICache/dead code optimization

10.1 Introduction

st200cc includes a binary phase optimization. This phase calls the **binopt** binary that performs ICache optimization, dead code and data removal.

st200cc provides a comprehensive set of options to perform ICache optimization. These options call the **binopt** binary, which may also be invoked directly as described in *Section 10.4: binopt on page 166*.

The ICache optimization options of the **st200cc** driver control the final function layout of the program. Function reordering in the final executable is important for targets with instruction cache. Indeed, functions that have temporal locality (determined by static analysis or profile based analysis), should also have spatial locality in order to optimize cache usage by reducing cache conflicts. Depending on the algorithm used for function placement, memory spatial locality or cache line spatial locality may be enforced.

Profile-based optimization (dynamic mode) is performed by first running the application, collecting profiling data and then recompiling and linking the application, passing in the profiling data to the ICache optimizer. Static analysis is performed by the optimizer using estimation instead of run-time data.

The **binopt** phase also performs dead code and unused data removal. **st200cc** provides a set of options to customize this phase.

Binary optimizations are performed in several phases, and it is important to pass the optimization options to both the compilation and the link phase. Several examples are given in *Section 10.3 on page 163*.

10.1.1 How it works

Binary optimizations are based on studying relocation at the assembler level.

The ICache optimizer builds a call graph (estimated or not according to the mode used). The edges are weighted by the number of calls between procedures. This weight is estimated from compiler information contained in the **.profile_info** section. Then, some algorithms stemming from the references are applied and an optimal function layout is found (see *Section 10.6: References on page 174*). Only functions whose body is in the **.text** section are candidates to move.

Dead code and dead data elimination follow the same principles. However, the generated 'relocation' graph takes into account all symbols (data, read-only data and code). This graph is traversed, and un-reached nodes are removed.

Candidate symbols to these optimizations must:

- have the **moveable** assembler attribute:
  - procedures beginning with **.proc** and ending with **.endp** automatically have the moveable attribute,
  - data needs the moveable attribute to be set with the following syntax:
    .type my_data, @object, moveable

Note: The compiler generates this attribute for symbols that it estimates to be moveable. This generation is not linked to any optimization level.

- have a size,
- be assembled with **--emit-all-relocs** option to **st200as**. This option is set by **st200cc** driver if dead code or ICache optimizations are required.

Such a symbol is called a **moveable** symbol in the rest of this section.

These conditions guarantee that all relocations within a moveable symbol are not resolved, and that all relocations pointing towards a moveable symbol are also not resolved. So, such a symbol can be safely removed or moved. Relocations will be resolved in a final link by **st200ld.**

*Note: 1 ICache optimization and dead code removal are implicitly turned on by the **st200cc** **-Os** and **-O2** (and higher) optimization options.*

2 Debug information is preserved by binary optimizations.

3 Mixing objects files compiled at different optimization levels is safe. Only symbols with the listed characteristics are candidates for binary optimization.

10.1.2 Synopsis

Several options specifically relate to ICache optimization and may be used in conjunction with the **--icache-opt** option, they are summarized below and described in detail in *Section 10.2 on page 156*.

- Enable or disable ICache optimization:

  st200cc --icache-opt=[on|off]

- Enable ICache optimization using a user-defined ordering file to order functions in the final executable:

  st200cc --icache-opt=on --icache-mapping=*file*.ly

- Apply compiler generated static analysis to the optimization:

  st200cc --icache-opt=on --icache-static
  [--icache-algo=*algorithm*]

- Apply profiling data (obtained during program execution) to the optimization:

  st200cc --icache-opt=on --icache-profile=*file*.dt
  --icache-profile-exe=*exe*.dt [--icache-algo=*algorithm*]

- Enable dead code and data removal:

  st200cc --deadcode

- Disable dead code and data removal:

  st200cc --no-deadcode

10.1.3 ICache files

Table 58 provides a synopsis of each of the files used in **binopt** optimization.

File	Description
`.dt`	Simulator trace file. This gives start and destination addresses for each jump in the execution sequence. Use the `TracePlugin` simulator plug-in. This has the extension `.so` for Unix hosts or `.dll` for Windows Hosts.[a] This file is passed to **st200cc** via the `--icache-profile` option.
`.ly`	Function layout file. This gives the order of functions in the final executable. This file is generated by the **st200cc** `--icache-static` or `--icache-profile` options when the `-keep` option of **st200cc** is used., or hand written and passed to **st200cc** with the `--icache-mapping` option.
`.po`	Pre-optimized object file. This file is a relocatable object file and is generated with the `-r` linker option. It contains unresolved relocations. A relocatable object file is used by the reordering phase (in order to move code, relocations must not be resolved). This file is kept when the `-keep` option of **st200cc** is used.
`.op`	Post-optimized object file. This file is a relocatable object. It contains reordered code. The final executable is obtained by resolving relocations in this file (simple link phase). This file is kept when the `-keep` option of **st200cc** is used.

Table 58: st200cc binopt optimization files

File	Description
`.icg`	Instruction cache call-graph file. This file is a user external file. Use this to add edges to the estimated static call-graph (see *Section 10.4.4: How to configure binopt on page 171* for more details).
`.rmlist`	Removed symbol file. This file is generated by **binopt** when dead code and data removal is activated. It contains list of symbols (data and functions) removed. It also contains some statistics about saved space. This file is generated when the `-keep` option of **st200cc** is used.

Table 58: st200cc binopt optimization files

a. To use the **`TracePlugin`** plug-in, for example to invoke **st200run** to run the simulator on a Unix host, use a command similar to the following:

```
st200run -d "st200sim 220 MODE ISS TRACING_ON true
TRACE_PLUGIN_MODULE
<tools-dir>/host/tplugins/TracePlugin
OUTPUT_TRACE_FILE file.dt" -- executable
```

<*tools-dir*> is the directory where the toolset is installed.

See the *ST200 Cross Development Manual* for further details about **st200run**.

10.2 st200cc options

The following key options are interpreted by **st200cc**:

`--icache-opt=`*`value`*

Enables or disables ICache optimization. ***`value`*** may be set to **`on`** for enable or **`off`** for disable. This option is implicitly turned on by the **st200cc** optimization options **`-O2`** or higher. All other ICache optimization options are controlled by this option.

`--icache-mapping`

Turns on function reordering based on a user specified ***`file`*`.ly`**. This is done in several steps:

- in the assembly phase the assembler is requested to emit all relocations,
- in link phase several steps are performed:
  - a pre-link phase generates a relocatable image of the executable (**`.po`** file),
  - a reordering phase [*BINOPT*] generates the optimized relocatable (**`.op`** file) from the input file (***`file`*`.ly`**) and the relocatable image,
  - a final link phase maps the optimized relocatable file to an executable.

--icache-static=*value*

Enables or disables static analysis based function reordering. ***value*** may be set to **on** for enable or **off** for disable. When this option is set to **on**, a new elf section (**.profile_info**) is created. This section contains frequency estimation of call counts. These sections are present only in **.po** and **.op** object file. The linker removes this section in the final link phase.

Static analysis is done in several steps.

- In the compiler back end, estimated block frequencies are emitted in a special section.
- In the assembly phase, the assembler is instructed to emit all relocations.
- The link phase is split into several phases.
  - A pre-link phase generates a relocatable image of the executable (**.po** file).
  - An optimization phase [*BINOPT*] generates the function layout file (**.ly** file) based on estimated block frequencies. The algorithm used for the computation of function layout can be changed with the **--icache-algo** option. Then, it generates an optimized relocatable image from the relocatable image and the function layout file (**.ly**).
  - A final link phase maps the optimized relocatable file to an executable.

--icache-profile=*file*.dt

Turns on profile based function layout. The ***file*.dt** drives the function placement algorithm. If ***file*.dt** does not exist, the optimization is skipped and the following message warning is emitted by the driver:

profile file not found, profile and relink your application

Profile based function layout has the following steps.

- In assembly the phase the assembler is instructed to emit all relocations.
- The link phase is split into several phases.
  - A pre-link phase generates a relocatable image of the executable.
  - An optimization and trace analysis phase [*BINOPT*] generates the function layout file based on the ***file*.dt**. This phase needs the original binary to build the symbol table and match addresses of the trace file with the function name. This binary is passed to this phase with the **--icache-profile-exe=*exe*** option of the **st200cc** driver. (If this option is omitted, the file specified with the **-o =*exe*** option to **st200cc** is used. In this case this file is overwritten by the final link phase.). If the ***exe*** file does not exist (or **a.out** if omitted), **--icache-profile** is ignored. Coherence between ***file*.dt** and the original binary ***exe*** is checked by the dates of these two files (***file*.dt** must be more recent than ***exe***). The algorithm used for the function layout can be changed with the **--icache-algo** option. Then, it generates an optimized relocatable image from the function layout file and the relocatable image.
  - A final link phase maps the optimized relocatable file to an executable.

--icache-algo=*algorithm*

Controls the placement algorithms used by static or profile based for ordering. ***algorithm*** may take the values **ph**, **col**, **trg** or **ltrg**.

- ***algorithm*=ph**

  This is the default. Enables a placement that optimizes main memory spatial locality of the code by implementing [**PH90**]. There is no code size impact with this algorithm.

- ***algorithm*=col**

  Optimizes main memory spatial locality with the knowledge of cache configuration to avoid cache conflicts. The implemented algorithm is [*HKC96*]. This algorithm may increase code size. It should be used for applications that have a large code covering / cache size ratio (> 2).

- ***algorithm*=trg**

  Enables a more optimal placement for minimization of cache conflicts at the cost of large code size increase. This implements the [*GBSC97*] algorithms and only works with profile based option. It uses temporal information from the profiling trace.

- ***algorithm*=ltrg**

  Enables a near optimal placement for minimization of cache conflicts at the cost of large code size increase. It should be used with caution or for experiments. It implements variations of the [*GBSC97*] algorithms. This algorithm might take a very long time to execute.

--deadcode/--no-deadcode

Activate/deactivate dead data and dead code removal.

Dead code and data removal is done in several steps.

- In the assembly phase, the assembler is instructed to emit all relocations.
- The link phase is split into several phases.
  - A pre-link phase generates a relocatable image of the executable (**.po** file).
  - An optimization phase [*BINOPT*] removes unused symbols and generates a lightened relocatable image.
  - A final link phase maps the optimized relocatable file to an executable.

10.2.1 Other options

All the **binopt** binary options may be passed to **st200cc** by prefixing them with **-Wo,**; see *Section 10.4: binopt on page 166*.

10.2.2 Default behavior

By default, only **moveable** procedures in the **.text** section are candidates to move. Code in user defined sections are not taken into account by **binopt**, however, a layout file can be defined for these procedures with the **--icache-mapping** option to **st200cc**.

However, since dead code and data removal is based on an analysis of the 'relocation' graph of the whole application, moveable symbols in any section are candidates for removal. In addition, **binopt** requires knowledge of the whole application, so all symbols have to be defined in the relocatable entry object (**.po** file).

10.2.3 Option combinations

If the **--icache-static**, **--icache-mapping** and **--icache-profile** options are used together on the same command line, each specified optimization is taken as far as possible while they do not compete. When they compete, **--icache-mapping** has first priority, **--icache-profile** has second priority and **--icache-static** has the lowest priority of the three options. If the file specified with **--icache-mapping** or **--icache-profile** does not exist, the option is ignored.

When combining ICache optimization and dead code and data removal, [*BINOPT*] phase makes both optimizations at once. Dead code and data removal is first executed, then ICache optimization is completed on the lightened code. This means that, dead functions do not pollute ICache optimization.

10.2.4 Full optimization

For the most efficient function layout optimization, the run-time libraries must be compiled with the **--icache-opt** and **--icache-static** options. Alternatively the global optimization option **-O2** (or higher), must be used. This applies to both the standard libraries provided with the **ST200** toolchain and to user libraries.

Dead code and data optimization also needs to compile libraries with **--deadcode** or **-O2** (or higher) options, because only code assembled by a special option of **st200as** (**--emit-all-relocs**) can be removed.

10.2.5 Relocatable files

The **-r** option to the linker, which can be used to generate a relocatable file is compatible with ICache optimizations. In this case, **-r** option is passed to the final link phase and the optimized relocatable object is generated.

The **-r** option deactivates the dead code and data removal optimization. Indeed, when building a relocatable object, **binopt** cannot see the whole application, some symbols may be undefined and resolved later.

10.2.6 Shared libraries, call shared executable

The dynamic mode (**-shared**, **-call-shared** options), which can be used to generate a shared library or an executable dynamically linked to shared libraries, is not compatible with binary optimizations. Binary optimization requires an intermediate relocatable object to be built which could not be later transformed into a dynamic object.

Note: Shared libraries and call-shared executables are not supported by this release.

10.2.7 Passing other options to the optimization phase

The **-Wo,[*option*]** can be used to pass specific options to the optimization phase [*BINOPT*].

For example:

- **-Wo,--sizecacheline,64 -Wo,--cacheline,512**
  These options set the configuration cache to 64 bytes x 512 lines.
- **-Wo,--begin,*my_main***
  These options activate the instruction cache optimization for ***my_main*** function only.

It is possible to enhance the estimated static call-graph with an external (**.icg**) file. This is only for optimizations based on a static analysis. To do this, use:

-Wo,--icg,*myconfig*.icg

A **.icg** file is a list of the following items:

source destination frequency

Where ***frequency*** is a float, and ***source*** and ***destination*** are function names.

A **.icg** file can be useful to enrich the estimated call graph with **syscalls** that are not detected via a simple relocation analysis. A default **.icg** file is taken. Use the following command to obtain the default file:

-Wo,--icgdump

See *Section 10.4.4: How to configure binopt on page 171* for more details.

The following option preserves a symbol and all its descendants during the dead code phase. More than one **--preserved** option may be given. Only one name may be given with each **--preserved** option. This symbol can be a function or data.

-Wo,--preserved,my_symbol

The following option preserves all symbols defined in **my_section** and in all its descendants. More than one **--section_preserved** option may be given. Only one name may be given with each **--section_preserved** option.

-Wo,--section_preserved,my_section

Likewise the **-Yo,*path*** options can be used to define a specific path for the [*BINOPT*] phase.

10.3 Examples

10.3.1 Compiling with static ICache optimizations

```
st200cc -O2 -o opt_exe *.c
```

or

```
st200cc --icache-opt=on --icache-static -o opt_exe *.c
```

or

```
# 2 phases
# compilation + frequency estimation (heuristic)
st200cc -O2 -c *.c
# link (repeat CFLAGS)
st200cc -O2 -o opt_exe *.o
```

or

```
# 2 phases
# compilation + frequency estimation (heuristic)
st200cc --icache-opt=on --icache-static -c *.c
# link (repeat CFLAGS)
st200cc --icache-opt=on --icache-static -o opt_exe *.o
```

10.3.2 Compiling a relocatable object with static ICache optimizations

This is useful for libraries.

```
# lib_stat is the main function of the library (algorithms
# use this function as the root of the static call-graph)
st200cc *.c --icache-opt=on --icache-static -o lib.o -v -Wl,-r
-Wo,--begin -Wo,lib_start
# use the archiver to produce the lib.a from lib.o
st200-ar -rc lib.a lib.o
```

or

```
st200cc *.c --icache-opt=on --icache-static -c
st200cc *.o --icache-opt=on --icache-static -o lib.o -Wl,-r -Wo,--begin
-Wo,lib_start
st200-ar -rc lib.a lib.o
```

10.3.3 To keep a generated layout file

```
st200cc --icache-opt=on --icache-static -o opt_exe -keep *.c -Wo,-v,-v
```

The generated file **opt_exe.ly** contains the computed layout file.

This layout can then be forced with:

```
st200cc --icache-opt=on --icache-mapping=opt_exe.ly -o opt_exe *.c
-Wo,-v,-v
```

10.3.4 Using profile driven ICache optimizations

```
st200cc --icache-opt=on -c *.c
```

First link:

```
st200cc --icache-opt=on -o exe *.o
```

Profile:

```
st200run  -d "st200sim 220 MODE ISS TRACING_ON true TRACE_PLUGIN_MODULE
<tools-dir>/host/tplugins/TracePlugin OUTPUT_TRACE_FILE trace.dt" -- exe
```

Re-link: **exe** must still exit

```
st200cc --icache-opt=on --icache-profile=trace.dt
--icache-profile-exe=exe -o opt_exe *.o -Wo,-v,-v
```

10.3.5 Using profile driven ICache optimizations using -O2 or higher

The compiler optimization options **-O2** or higher are used.

```
# Build first executable with instruction cache information
# (default behavior with -O2 or upper)
st200cc *.c -O2 -c
st200cc *.o -O2 -o exe --no-deadcode
```

Profile:

```
st200run  -d "st200sim 220 MODE ISS TRACING_ON true TRACE_PLUGIN_MODULE
<tools-dir>/host/tplugins/TracePlugin OUTPUT_TRACE_FILE trace.dt" -- exe
```

Re-link with the profile file:

```
st200cc *.o -O2 --icache-profile=trace.dt --icache-profile-exe=exe -o
opt_exe -Wo,-v,-v
```

10.3.6 Compiling with st200gprof file-driven ICache optimizations

Compile using profiling:

```
st200cc *.c -O2 -keep -o exe -pg --no-deadcode
```

Execute and create the **gmon.out.000** file:

```
st200run -d "st200sim 220 MODE ISS" -- exe
```

Recompile source file without profiling:

```
st200cc *.c -O2 --icache-profile=gmon.out.000 --icache-profile-exe=exe
-keep -o opt_exe -Wo,-v,-v
```

10.3.7 Linker options

The majority of arguments intended for linker phase (such as **-Wl,-*[option]***) are passed to the first relocatable link phase (when this phase is needed). However some specific options are only passed to the final link phase (after [*BINOPT*] phase): **-T**, **-Wl,-T**, **-Wl,-r**, **-r**, **-Wl,--relax**, **--relax**, **-Wl,-Map**, **-Wl,-M**.

Some others linker options deactivate binary optimization:
-Wl,-shared, -Wl,--shared, -Wl,-dy, -Wl,-Bdynamic, -Wl,-call_shared, -Wl,-export-dynamic, -Wl,-E, -Wl,-dynamic-linker, -Wl,--dy, -Wl,--Bdynamic, -Wl,--call_shared, -Wl,--export-dynamic, -Wl,--dynamic-linker, -call_shared, -shared.

10.4 binopt

This section describes how to invoke and use the **binopt** binary to perform instruction cache optimization.

*Note: The options to **binopt** are accessible through the **st200cc** driver, by prefixing them with the **-Wo** option.*

10.4.1 Synopsis

<*tools-dir*>/bin/binopt [options] --input <*file-in*> --output <*file-out*>

where **<*tools-dir*>** is the directory the toolset is installed in, ***file-in*** is an input executable (relocatable binary for static mode, real executable for dynamic mode).

10.4.2 Options

Global binopt options

--icacheopt Perform instruction cache optimizations.

--deadcode Perform dead code elimination.

--codereorder Perform code reordering from a hand-written layout file.

--input Input object file (relocatable object).

--output Output object file (relocatable object).

--input_exe Input executable (real executable for dynamic ICache optimization).

--deadcodeobject <*file*> Keep intermediary output object file after dead code elimination.

--layoutfile <*file*> Keep intermediary output layout file of functions (**.ly** file).

--deadcodefile <*file*> Keep intermediary list of removed symbols.

--preserved <*name*> Preserve a symbol and all its descendants. More than one **--preserved** option may be given. Only one name may be given with each **--preserved** option.

--section_preserved <*my_section*>

Preserve all symbols defined in ***my_section*** and all its descendants. More than one **--section_preserved** option may be given. Only one name may be given with each **--section_preserved** option.

--scan_end_proc (-E) Use old **$endproc** label to find function size (default is **false**, function size is now deduced from elf attributes).

User command file

@<*file*> Give a user command file. See *Section 10.4.5: User command file on page 172*.

Input and output options

--icg (-g) <*file*> ICache call-graph user configuration (user defined edges). See *Section 10.4.4: How to configure binopt on page 171*.

--outlayout (-o) <*file*> Specify file where layout is written (if omitted output is sent to **stdout**).

Trace options

--davinci (-y) <*file*> Dump call-graph into davinci syntax. See *daVinci V2.1 dump on page 170*.

--dump (-d) Dump call-graph to **stdout** in xvcg format. See *VCG/XVCG dump on page 170*.

--icgdump (-D) Dump ICache call-graph user configuration.

--help (-h) Displays a help page of options.

--version (-V) Show version.

--verbose (-v) Increase verbose level by one (default is 0).

Type of optimization

--coloring (-C) Change the placement algorithm to coloring algorithm [*COL*].

--ltrg (-L) Change the placement algorithm to Large Temporal Relationship Graph [*LTRG*] (only applicable in profiling mode and may be very long).

--ph (-P) Change the placement algorithm to Pettis & Hansen [*PH90*] (default algorithm).

--ph_col (-O) Change the placement algorithm to a mix of [*PH90*]and coloring [*PH_COL*]. Use [*PH90*] call-graph merge with [*COL*] node merge. Node merge attempts to minimize cache conflicts using the set of the available colors for each procedure.

--profile (-X) Perform dynamic optimization from a profile file (default mode).

--trg (-T) Change the placement algorithm to Temporal Relationship Graph [*TRG*] (only applicable in profiling mode).

-- static (-S) Perform static optimization from **.profile_info** section contents.

call-graph configuration

--begin (-b) <*proc*:*weight*>

Start symbol with its weight (default is **main:1.0**). Several **-b** options are allowed. Weight is only taken into account in static mode.

--ignore (-i) <*proc*> Ignore symbol (default is empty). Several **-i** options are allowed.

See *Section 10.4.4: How to configure binopt on page 171*.

Algorithm specific options

--cacheline (-c) <*nb*> Number of lines in the cache (default is **512**), for all algorithms except for [*PH90*].

--chunksize (-s) <*size*> Chunk size (default is **64**). Each function is split into chunks of size **<*size*>**, only for [*TRG*] and [*LTRG*].

--fetch (-f) <*size*> Size of “addend to size” in cache line, to take into account prefetch (default is **1**), only for [*COL*] and [*PH_COL*].

--sizecacheline (-z) <*size*>

Size of a cache line in bytes (default is **64**), for all algorithms except for [*PH90*].

call-graph specific options

--deffreq (-e)	Default call site frequency (default 1.000000e+00), for non instrumented part of code, only for static mode.
--prune (-p) <*float*>	Percentage of the sum of edge-weights kept for pruning graph (default 100 - 1.000000e-07), only for static mode.
--trace (-t) <*file*>	Branch sampling trace file (syntax: ***src dest nb_of_calls***), only for profile mode.

10.4.3 binopt dump options

binopt enables a call-graph to be generated either a static call-graph, or a profile call-graph. Within the call-graph “nodes” are procedures and “weight of the edges” is the number of calls (estimated for a static call-graph or based on the profile trace).

VCG/XVCG dump

By using the **--dump** option, **binopt** generates a XVCG (Visualization of Compiler Graphs) file which is sent to **stdout**.

Example:

```
binopt -o myexe.ly --static --icacheopt --input myexe.po --dump
--output myexe.op > myexe.dump
```

or

```
st200cc *.c -O2 -Wo,--dump -o myexe > myexe.dump
```

then

```
xvcg myexe.dump&
```

VCG/XVCG - USAAR Visualization Tool V.1.3 can be downloaded at:
http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html#availability

daVinci V2.1 dump

When the **--davinci <*file*>** option is used, **binopt** generates a daVinci file in **<*file*>**.

Example:

```
binopt -o myexe.ly --static --icacheopt --input myexe.po --davinci
myexe.davinci --output myexe.op
```

or

```
st200cc *.c -O2 -Wo,--davinci,myexe.davinci -o myexe
```

then

```
daVinci myexe.davinci&
```

daVinci V2.1 can be downloaded at: http://www.tzi.de/~davinci

10.4.4 How to configure binopt

To help **binopt** to better estimate the call-graph of your final application, a set of options are provided.

- **-icg <*file*>**, ICache call-graph user configuration file.

  In this file, it is possible to add or remove edges of the estimated static call-graphs. Use **binopt --icgdump** to see an example of such a file. It can be useful to add edges which are not caught by a simple relocation analysis (indirect call or syscalls), or to remove polluting calls which you know are not taken. This option is only valid for static mode.

- **-ignore <*proc*>**, all calls to ***proc*** and its descendants are ignored.

  It may be useful to ignore calls to a library procedure. This option may be used in both static and profile mode. For example:

  ```
  --ignore printf --ignore fprintf
  ```

- **-begin <*proc*[:*weight*]>**, any procedure specified with this option is interpreted as an entry point in your application.

  Relative weight can be specified to give more importance to one procedure in relation to another. This option can be used in both static and profile mode. In profile mode, weight is ignored. For example:

  ```
  --begin myapp_init:0.3 --begin myapp_main:0.5 --begin
  myapp_finish:0.2
  ```

  or in profile mode:

  ```
  --begin myapp_init --begin myapp_main --begin myapp_finish
  ```

10.4.5 User command file

A command file containing options to pass to **binopt** is specified by using the **@<*filename*>** option. Options can be separated by space, tabulation or newline.

For example:

```
cat cmd.binopt
       >--begin init:0.2 --begin main:0.8
       >--ignore printf
       >--ignore fprintf
       >--davinci myapp.davinci

binopt --layoutfile myexe.ly --static --input myexe.po @cmd.binopt
--output myexe.op --icacheopt
```

or

```
st200cc *.c -O2 -Wo,@cmd.binopt -o myexe
```

10.5 Warning and error messages

10.5.1 binopt

- “symbol %s is not a function: ignored”: warning: Symbol defined in an **.icg** file is not a function.
- “no need of trace file in static mode”: error: use **--trace** option with **--static**.
- “No .profile_info section present. BB frequency is 1.”: warning: there are not any **.profile_info** section in the input object file whereas **--static** is used. **binopt** estimated all call counts to 1.
- “Sorry too many start functions max is %d”: error: user gives more than 128 start functions.
- “Sorry too many ignored functions max is %d”: error: user gives more than 128 ignored functions.
- “unknown argument %s”: error: wrong option.
- “Too many arguments”: error: too many arguments on the command line.
- “need executable”: error: input object file is not given to **binopt**.
- “fatal error opening file %s”: error: cannot access input object file.

- “Not a bfd_object !”: error: input object file is not recognized.
- “ERROR:Unable to find any calls in the trace file %s!!!”: error: in profile mode, cannot find any calls.
- “need trace file”: error: trace file has been omitted.
- “Can't build graph for daVinci with this algorithm”: in profile mode, daVinci dump is not accessible for TRG and LTRG algorithms.
- “start symbol %s is not a function: ignored”: warning: start symbol given with **`--begin`** option is not a function.
- “start symbol %s is not global: ignored”: warning: start symbol given with **`--begin`** option is static.
- “ignored symbol %s is not a function: ignored”: warning: ignored symbol given with **`--ignore`** option is not a function.
- “ignored symbol %s is not global: ignored”: warning: ignored symbol given with **`--ignore`** option is static.
- “overflow during normalization of static call-graph”: warning: during the normalization of static call-graph, overflow has been detected. A re-normalization is done to avoid overflow in next phases.
- “Can't apply this algorithm placement for static estimation, take PH.”: warning: TRG or LTRG algorithm are not accessible in static mode.
- “Cannot find %s in symbol table\n”: warning: a symbol given in a **`.ly`** input file cannot be found in the symbol table.
- “Cannot find %s with index %d”: warning: warning: a symbol given in a **`.ly`** input file cannot be found in the symbol table with this index (case of several static functions with the same name).
- “Can't open link order file %s”: warning: input ordering file (**`.ly`**) cannot be accessed. No reordering is done, just copying.
- “bfd error %s”: error: input object file is not a bfd object.
- “%s: No symbols”: warning: cannot find any symbol in the input object file.
- “Cannot remap symbol %s as requested in the link mapping: symbol not found, with unknown size or already placed”: warning: a function cannot be moved.
- “input mapping file %s is in wrong format”: warning: syntax error in the reordering input file (**`.ly`**). Ignore this file, output object file is a copy of input.
- “no relocation found just copying”: warning: cannot find any relocation, so reordering is impossible. Output object file is a copy of input.

10.6 References

[*COL*]: “Efficient Procedure Mapping Using Cache Line Coloring”, Hashemi, Kaeli & Calder (1996) WRL Research Report 96/3.

[*GBSC97*]: “Procedure Placement Using Temporal Ordering Information”, NiKolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder, IEEE 1997.

[*HKL96*]: “Efficient Procedure Mapping using Cache Line Coloring”, Air H, Hashemi, David R. Kaeli and Brad Calder, WRL Research Report 96/3, 1996.

[*BINOPT*]: Optimization at binary level. Binary name is **binopt**. This utility performs instruction cache optimization, dead code and data removal. It includes old **codereorder** and **icacheopt** tools.

[*LTRG*] An algorithm inspired from [*TRG*], using a multi-graph that represents exactly cache conflicts.

[*PH90*]: “Profile Guided Code Positioning”, Karl Pettis and Robert c. Hansen, ACM 1990.

[*PH_COL*]: An algorithm inspired from [*PH90*] for graph merge and [*COL*] for graph node union (take into account colors to minimize conflict, insert blank when needed).

[*TRG*] “Procedure Placement Using Temporal Ordering Information”, Gloy, Blackwell, Smith, Calder (1997) published in the Proceedings of Micro-30.

11 Assembler

11.1 Introduction

This chapter documents the assembler and the assembly language for the ST200 processor family.

The ST200 assembler is based upon **gas**, the GNU assembler. Much of the information about assembler directives and expression syntax is repeated, in abbreviated form, from the standard **gas** documentation. The primary areas in which the ST200 assembler deviates from **gas** are in the syntax for instructions and in the restrictions placed upon the standard directives. In addition, the ST200 assembler accepts a number of ST200 specific directives and options. The description of ST200 instruction syntax assumes some familiarity with the architecture of the ST200 processor family.

Section 11.2 details the syntax of assembler source programs.

Section 11.3 describes how the assembler is run: the command line options, input and output file formats.

11.2 Assembler syntax

11.2.1 General description

An ST200 assembly file consists of three types of fields: comments, directives and instructions. Comments are treated by the assembler as white space and, other than syntax, have little bearing on the subsequent description.

Directives include constant declarations, memory allocation commands, and commands to guide both the assembler and linker.

Instructions in the ST200 architecture may consist of multiple operations that are to be executed simultaneously. In the following discussion, we use the term **bundle** to refer to a group of operations that are defined to be executed simultaneously.

An assembly file may consist of an arbitrary interleaving of directives, comments, and bundles (with a few restrictions on specific directives). A bundle consists of a sequence of operations and comments terminated by a pair of semicolons "**;;**" on a separate line.

11.2.2 Comments

There are two types of comments supported by the assembler. In both cases the comment is equivalent to one space. The input routines for the assembler, filter out all comments and replace them with a single space.

- C-style comments consist of an text beginning **/*** and ending with the next ***/**. C-style comments may not be nested.
- Line comments consist of any text on a single line beginning with the first pound character, **#**. A line comment can appear on a line by itself or as the last thing on any input line.

To be compatible with past assemblers, a special interpretation is given to lines that begin with **#** followed by an absolute expression. In this case the expression is interpreted as the logical line number of the next line. An optional string following such an expression is interpreted as a new logical file name. The rest of the line, if any, should be whitespace.

Example:

```
# This is an ordinary comment.
# 42-6 new_file_name  # New logical file name
                      # This is logical line #36.
```

Comments beginning with the strings **#APP** or **#NO_APP** also have a special interpretation and should not be used.

11.2.3 Lexical categories

11.2.3.1 Whitespace

Whitespace consists of one or more blanks or tabs, in any order. Unless whitespace occurs within character constants, any whitespace sequence is treated as exactly one space.

11.2.3.2 Symbols

A symbol is one or more characters chosen from the set of all letters (both upper and lower case), digits and the four characters **_.$?**. No symbol may begin with a digit. Case is significant. All characters are significant. Symbols are delimited by characters not in the set described above, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter).

11.2.3.3 Constants

A constant is a number, written so that its value is known by inspection, without knowing any context.

Example:

```
.byte  74, 0112, 0x4A, 0X4a, 'J, 'J'   # All the same value
```

- **Character constants**. There are two kinds of character constants. A **character** stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string **literals**) are potentially many bytes and their values may not be used in arithmetic expressions.
- **Strings**. A **string** is written between double-quotes. It may not contain double-quotes or null characters. Special characters may be included in a string by using escape sequences as with C-language programs. Strings may be automatically null terminated in some contexts (**.string**, **.asciiz**).

- **Numerical constants**. The assembler distinguishes three types of numbers according to their storage requirements in the target machine. **Integers** are numbers that fit into 32 bits (an "**int**" in the C language). **Bignums** are integers, but they are stored in more than 32 bits.

  - **Integers**.

    A binary integer is "0b" or "0B" followed by zero or more of the binary digits, either 0 or 1.

    An octal integer is "0" followed by zero or more of the octal digits chosen from 01234567.

    A decimal integer starts with a non-zero digit followed by zero or more digits chosen from 1234567890.

    A hexadecimal integer is "0x" or "0X" followed by one or more hexadecimal digits chosen from 0123456789abcdefABCDEF.

  - **Bignums**.

    A bignum has the same syntax and semantics as an integer except that the number (or its negative) may require more than 32 bits. Bignums are not necessarily permitted in all places where integers are permitted.

  - **Floats**.

    The ST200 floating-point runtime model defines two of the IEEE-754 standard data type formats, single and double. The IEEE single format has a precision of 24 bits (24 significant bits), and 32 bits overall. The IEEE double format has a precision of 53 bits, and 64 bits overall.

    .float <*expression*>
    .double <*expression*>

    Where **<*expression*>** should have the format:

    0e:<*hex digits*>

    The **<*hex digits*>** are the hexadecimal form of the IEEE representation (32-bit for **.float**, 64-bit for **.double**).

    It is not possible to use true floating-point format (for example: **+1.0e+10**), because this is not currently implemented in the assembler.

    .real4 <expression>:<number>
    .real8 <expression>:<number>

    Where **<*expression*>** is an integer or hexadecimal constant, just like **.data4** or **.data8**. The integer or hexadecimal constant is the binary representation of the IEEE format (32-bit for **.real4**, 64-bit for **.real8**).

11.2.3.4 Statements

A statement begins with zero or more labels, optionally followed by a key symbol which determines the type of the statement. The key symbol determines the syntax of the rest of the statement. Assembler directives begin with a "**.**". The directives accepted by the assembler are described in *Section 11.2.9: Assembler directives on page 187*. Any statement that begins with a letter is part of an ST200 instruction bundle. The bundle terminator (**;;**) is also a statement.

A statement is terminated by a newline character. Newlines within character constants are an exception, they do not terminate a statement. It is an error to end any statement with end-of-file; the last character of any input file must be a newline. Statements may be written on multiple lines provided that each line other than the last ends with a backslash (****).

11.2.3.5 Labels

A label is a symbol immediately followed by one or two colons ("**:**" or "**::**"). Whitespace before a label or after the colon(s) is permitted, but whitespace between the symbol and colon(s) of a label is not permitted. Labels followed by a single colon are treated as local symbols, labels followed by a double colon are global symbols - they are visible to other object modules linked with the output of the assembler. Symbols can also be marked as global with the **.global** directive.

11.2.4 Symbols

The lexical specification for symbols has already been described. This section defines the semantics of symbols.

11.2.4.1 Giving symbols other values

A symbol can be given an arbitrary value by writing a symbol followed by a double equals sign (**==**), followed by an expression. This is equivalent to using the **.equ** or **.set** directives discussed later. Symbols are not required to be defined before use; however, the current implementation of the assembler allocates space for long immediates whenever a symbol is undefined even if the symbol is eventually resolved to a value that can fit in a short immediate.

Example:

```
foo == 3
```

11.2.4.2 Local and global labels

Labels defined with a double colon are global, that is they are visible to all the subprograms being linked. All other labels are local, that is they are only visible in the source file where they are declared.

Example:

```
local_label:
global_label::
```

11.2.4.3 Symbol names

Symbol names are required to begin with a letter or one of "**_?**". The initial character may be followed by any string of digits, letters and characters from the set "**._?$**". The case of letters is significant: **foo** is a different symbol from **Foo**.

- Local symbol names. Local symbols may be defined for use in a given source file, but are not represented in the object file for subsequent use by a linker or debugger. By convention, local symbol names begin. "**.L**" (upper case), "**L?**","**$**", "**?**", or "**_?**". The assembler writes these symbols to the object file if the "**-L**" option is used.
- The dot symbol. The special symbol "**.**" refers to the current address that is being assembled. Therefore the expression "***here*: .long .**" defines the location ***here*** to contain its own address. Assigning a value to "**.**" is treated the same as a "**.org**" directive. This means that the statement "**. == . + 4**" is the equivalent to "**.skip 4**" discussed later.
- Value. The value of a symbol is defined as a 32 bit 2's complement number. For a symbol which labels a location in the text, data, or bss sections, the value is the offset (in bytes) from the start of the section to the label.

11.2.5 Expressions

An **expression** specifies an address or numeric value. An expression consists of one or more **arguments** delimited by **operators**. The arguments of an expression are symbols, numbers or sub-expressions.

11.2.5.1 Sub-expressions

A sub-expression is either an expression within parentheses or a prefix operator followed by an argument.

Example:

```
( 3 + foo )
-3
```

11.2.5.2 Absolute expressions

An expression that can be evaluated to a value without knowledge of addresses (one that is not relative to the start of some object section) is called absolute. Expressions that are not absolute have the form (**section + offset**) where **section** is typically one of text, data, bss or undefined, and **offset** is an offset into the section (a signed, 2's complement, 32 bit integer). Some expression operators may only be applied to absolute sub-expressions.

11.2.5.3 Operators

- Prefix operators. The assembler supports the following unary prefix operators. The argument must be absolute.

  `-` Negation. Two's complement negation.

  `~` Complement. Bitwise not.

- Infix operators. The assembler provides a number of binary infix operators. Apart from "`+`" or "`-`", both arguments to the operator must be absolute; the result is absolute. Within a precedence class, operators are applied left to right.

  - Highest precedence.

    `*` Multiplication.

    `/` Division: truncation is the same as the C operator "/".

    `%` Remainder.

    `<` Relational less.

<< Shift left, same as the C operator “<<”.

> Relational greater.

>> Shift right, same as the C operator “>>”.

- Intermediate precedence.

| Bitwise inclusive OR.

& Bitwise AND.

^ Bitwise exclusive OR.

! Bitwise NOT.

- Lowest precedence.

+ Addition.

- Subtraction.

11.2.6 Bundles

An ST200 processor may issue multiple conventional instructions (operations) in a single cycle; these operations are statically scheduled. A group of simultaneously issued operations is called a bundle. The end of a bundle is denoted by “**;;**”. Each operation of a bundle, as well as the bundle terminator must reside on a separate line in the source file (each operation is treated as a separate “statement”). The bundle may be interleaved with comments. Each operation of a bundle is a separate syntactic “statement”.

Example:

```
operation0 # a comment
operation1 /* another comment */
operation2
;;
```

Labels may not appear within a bundle; they must occur before the first operation of the bundle. The destination of a branch must be the start of a bundle.

The ST200 processors are implemented with a single cluster called **c0**. The assembler uses the operand syntax to determine the target cluster for a particular operation; however, as an aid to readability, a cluster identifier may optionally precede each operation of a bundle. These identifiers have the form (**cx**) where ***x*** is a cluster number.

Example:

```
c0 operation0
operation1
;;
```

The assembler does not interpret the cluster identifiers, they are purely an aid to reading complex code. In order to determine where a syllable is scheduled (and hence to generate the bits corresponding to a bundle) the assembler uses cluster information embedded in register names. The syntax for operands is discussed in *Section 11.2.7*.

11.2.7 Operands

There are two basic types of operands used in defining the ST200 operations: register identifiers and immediates.

11.2.7.1 Registers

The syntax for a register descriptor is:

```
$ register_type   cluster.register_number
```

where:

- the register type is a string (usually one character) that defines the bank of registers within a cluster,
- the cluster equals 0,
- the register number denotes the register within that bank.

Example:

```
$r0.1     general purpose register 1 of cluster 0
$b0.3     branch register 3 of cluster 0

c0  add $r0.3   = $r0.3, 42 # r0.3  <- r0.3 + 42
c0  add $r0.11  = $r0.12, 1 # r0.11 <- r0.12 + 1
;;
```

In the preceding example, two adds are scheduled to be issued in a single cycle. As can be seen, the cluster identifiers are redundant.

In the currently planned ST200 processors, **c0** has the following register banks:

$r A bank of 64 general purpose 32 bit registers.

$b A bank of 8 single bit branch (or condition).

The branch registers are used as condition bits for branch and select operations as well as operations that generate or consume carry bits.

There are a few general purpose registers with defined functions:

$r0.0 These registers always return 0 when read.

$r0.63 This register, known as the link register, is used to store the return address for all call operations. In addition, it is read for the target address by some call and goto operations.

To simplify the ST200 assembly syntax, starting from the R3.2 release of the toolchain, the cluster number is no longer required in the register syntax. By default, the cluster '0' will be used by the assembler. The two following instructions are then equivalent:

```
add $r11   = $r12  , $r2
add $r0.11 = $r0.12, $r0.2
```

11.2.7.2 Immediates

There are two basic types of immediates – absolute and PC relative. In the ST200 family, PC relative immediates are used for **branch**, **call**, and **goto** operations while all other immediates are absolute.

All immediates are encoded as 2's complement numbers. Immediates that can be represented with a small number of bits, are encoded directly in the machine word containing the operation, while longer immediates require an extension word.

The ST200 assembler allows all immediates to be defined by a number, symbol, or expression. In the case of PC relative immediates defined by labels, the assembler calculates the value relative to the PC.

11.2.8 Operations

There are two basic types of operation statements accepted by the assembler: integer operations and control flow operations such as branches. The syntax of each class of operation is described in detail below. Case is significant in operation names: all operation names are lower case.

11.2.8.1 Integer operations

As a general rule, operands that are written by an operation appear to the left of any "**=**", while operands that are read appear to the right. When more than one operand is read (written), the operands are separated by "**,**". There are a small number of operations (branch and prefetch operations) that do not write an operand and hence have no "**=**". The optional prefix **c0** denotes cluster 0.

Example:

```
c0 slct $r11 = $b2, $r3 , 3
   sub $r2   = 3, $r4
```

11.2.8.2 Conditional branches

Conditional branch operations read a branch register (**$b0.y**) and conditionally branch to a destination.

Example:

```
c0 br $b0.1, dest_label
```

In this example, the branch is taken when **$b0.1** contains 1.

Branch registers, which are read by conditional branch operations, are written by other operations including compares and arithmetic operations that generate a carry. These branch registers are also read by "select" operations and arithmetic operations that read a carry bit.

11.2.8.3 Call, goto and return

call and **goto** operations are similar to branches, except that they are unconditional. Each of these operations has two forms, an immediate, PC relative, form, and an indirect form that uses the system link register (**$r63**). Furthermore, the **call** operations write the link register with the address of the bundle following the current bundle. The **return** operation is a pseudo-operation for the indirect form of the **goto**. In order to make all the state used by an operation explicit, these operations explicitly name the link register even when it is implicit.

```
call $r63 = $r63              # call through the link register
goto dest_label               # goto a destination
return $r63                   # same as goto $r63
```

Only one **branch**, **goto**, or **call** operation may be issued in a single bundle. Furthermore, the bundle sequentially following a taken **branch**, **goto**, or **call** operation is quashed.

11.2.8.4 Memory operations

The ST200 processors support three basic categories of memory operations: loads, stores, and prefetches. In the current processors, a bundle may include, at most, one memory operation.

Memory addressing

The ST200 architecture only supports one addressing mode. All memory accesses are made via a pointer in a register plus an immediate offset. The assembler syntax for this is:

```
offset [ register ]
```

Example:

```
stw 4[$r1] = $r23  # *($r1 + 4) <- $r23
```

Dismissible modifiers

The ST200 processors support speculative (dismissible) load operations. In particular it is possible to provide code that dereferences a pointer prior to the test to determine that the pointer is valid. A memory operation may be defined by software to be dismissible, in the event that a non-recoverable virtual memory fault occurs by performing a dismissible load, the value returned is defined to be 0. A dismissible load is denoted by the operation modifier "**.d**". In the following example, **probe(a)** evaluates to true if **a** can legally be dereferenced.

Example:

```
ldw.d $r2 = 0[$r1]  # $r2 <- probe($r1) ? *($r1): 0
```

Prefetch

The ST200 processors provide a group of prefetch operations that bring data into the appropriate cache, but do not modify a register and do not stall the machine on a cache miss.

Example:

```
pft   8[$r8]    # fetch the cache line containing * (8+$r8)
```

11.2.9 Assembler directives

The ST200 assembler supports most of the standard directives provided by the GNU assembler. Only a subset of the directives accepted by the assembler is described here.

11.2.9.1 Alignment directives

At present, the directives that insert values into memory, or change the location counter, are only valid in the data section. Using them in the text section is reported as an error.

`.align abs-expr, abs-expr`

This pads the location counter, in the current section, to the specified boundary. The first expression (which must be absolute) is the alignment request in bytes. The alignment request must be a power of 2. The second, optional, expression (also absolute) gives the value to be stored in the padding bytes. The default case is to pad by 0.

Example:

```
.align 8
```

The preceding advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is made.

11.2.9.2 Section directives

`.section` ***`name`***	Assemble the following statements into section ***`name`***.
`.data`	Assemble the following statements into the data section.
`.text`	Assemble the following statements into the text section.

11.2.9.3 Data directives

.comm *symbol, length, alignment*

Declare a named common section in the bss section. The ***length*** and ***alignment*** parameters must be absolute expressions.

.skip *size, fill*

Fill the next ***size*** bytes with the optional ***fill*** value. The default case is to fill with 0.

.byte *expression:number, ...*
.data1 *expression:number, ...*

These directives, which are equivalent, take one or more pairs of parameters separated by commas. The first value of the first expression is assembled into the next byte location. The second, optional, parameter indicates the number of duplicate bytes to generate. Any expression that evaluates to a value that cannot be represented by a byte is truncated with a warning.

.hword *expression:number, ...*
.data2 *expression:number, ...*

These directives, which are equivalent, take one or more pairs of parameters separated by commas. They assemble the value of the first expression into the next two bytes. If the expression evaluates to a value that requires more than 16 bits to represent it then it is truncated and a warning issued. The second parameter in each pair is optional; if present it specifies the number of consecutive 16 bit locations to fill with the value of the first parameter.

.word *expression:number, ...*
.data4 *expression:number, ...*
.real4 *expression:number, ...*

These directives, which are equivalent, take one or more pairs of parameters separated by commas. They assemble the value of the first expression into the next four bytes. The second parameter in each pair is optional; if present it specifies the number of consecutive 32 bit words to fill with the value of the first parameter. It is expected that when **.real** is used, the expression represents a floating point number.

```
.quad  expression:number, ...
.data8 expression:number, ...
.real8 expression:number, ...
```

These directives, which are equivalent, take one or more pairs of parameters separated by commas. They assemble the value of the first expression into the next eight bytes. The second parameter in each pair is optional; if present it specifies the number of consecutive 64 bit words to fill with the value of the first parameter.

11.2.9.4 Symbol directives

```
.equ symbol, expression
.set symbol, expression
```

These directives, which are equivalent, set the value of ***symbol*** to ***expression***. This changes the symbol's value and type to conform to that of expression. If the symbol was flagged as external (see **.global**) then it remains external. The value of a symbol can be set many times in the assembly file. If the symbol is external, then the value stored in the object file is the last value assigned to it.

```
.global symbol
```

This directive defines a symbol as external (that is, visible to the linker). A label can also be marked as global directly by defining it with a double colon,"**::**".

11.2.9.5 Include directive

```
.include file
```

This directive provides a way to include supporting files at specified points in your source program. The code from "***file***" is assembled as if it followed the point of the **.include**; when the end of the included file is reached, assembly of the original file continues.

11.2.9.6 ST200 machine directives

.assume *parameters*

The **.assume** directive expects one or more parameters, separated by commas. It is also possible to use several consecutive **.assume** directives.

The parameters of the **.assume** directive are encoded in the elf flags of the object file to identify the basic code generation assumptions taken by the compiler. This allows the linker to detect attempts to link code assembled with different incompatible assumptions.

The **.assume** directive is generated by the C compiler. The user should only consider writing a **.assume** directive when mixing assembly code with C code.

The **.assume** directive accepts five types of parameters, listed in *Table 59*. Values within the parameter types are exclusive.

Parameter type	Values	Description
Core identification	**st220**, **st231**	Defaults to **st220** if not already specified in the assembly file or by the **-mcore** command line option. **Note:** this directive does not impact the encoding table used by the assembler, only the ELF flags.
Silicon implementation identification	**cut0**, **cut1**, **cut2**, **cut3**, **cut4**, **cut5**	Defaults to **cut0** if not already specified in the assembly file.
Software ABI convention	**no-abi**, **old-multiflow-abi**, **lx-embedded-abi**, **pic-abi**, **gcc-abi**	Defaults to **no-abi** if not already specified in the assembly file.
OS ABI identification	**bare-machine**, **os21**, **linux**	Defaults to **bare-machine** if not already specified in the assembly file.

Table 59: .assume directive parameters

.proc *parameters*

This directive is supported for compatibility reasons with existing code. The parameters are ignored.

.endproc *parameters*

This directive is supported for compatibility reasons with existing code. The parameters are ignored.

.type *symbol*, [@object | @function], moveable

Declare the symbol to be either the name of an object or a function. If moveable is specified, it can be moved around by the ICache optimizer.

11.2.9.7 ST200 special directives

The following directives are generated by the compiler, but are ignored by the assembler. In most cases they are used by the compiled simulator.

.__longjmp, **.__setjmp**, **._longjmp**, **._setjmp**, **.call**, **.comment**, **.entry**, **.import**, **.return**, **.sversion**, **.type**

11.2.9.8 Restrictions on standard directives

A number of restrictions currently apply to standard directives.

The following directives are not currently permitted in the text section:

.ascii, **.asciz**, **.byte**, **.data1**, **.data2**, **.short**, **.string**

The following directives are permitted in the bss section, but they only have an impact on the size of the bss section, not its content (because the bss section has no content in the binary file):

.ascii, **.asciz**, **.byte**, **.data1**, **.data2**, **.short**, **.string**, **.data4**, **.data8**, **.hword**, **.long**, **.octa**, **.quad**, **.real4**, **.word**, **.single**, **.double**, **.float**

The following directives are currently ignored:

.dc, **.dc.b**, **.dc.d**, **.dc.l**, **.dc.s**, **.dc.w**, **.dc.x**, **.dcb**, **.dcb.b**, **.dcb.d**, **.dcd.l**, **.dcb.s**, **.dcb.w**, **.dcb.x**, **.ds**, **.ds.b**, **.ds.d**, **.ds.l**, **.ds.p**, **.ds.s**, **.ds.w**,**.ds.x**, **.fill**, **.lflags**, **.mri**, **.org**, **.p2align**, **.p2alignw**, **.p2alignl**

11.3 Invoking the assembler

11.3.1 Assembler command line

Here is a a brief summary of how to invoke **st200as**:

```
st200as [ -a[hls][=file] ] [ -EB ] [ -EL ] [ -I dir ] [ -J ]
[ -L ] [ --mcore=st220|st231 ] [ -o objectfile ] [ -R ]
[ --statistics ] [ -v|--verbose ] [ --version ] [ -version ]
[ -W ] [ -Z ] [ -- | files... ]
```

The assembler can also be invoked by running the C compiler, with an input file that has a "**.S**" suffix. This allows the C pre-processor to be used. See *Chapter 2: st200cc on page 9* for details.

11.3.1.1 Command line options

This section gives a more detailed description of the command line options of the assembler.

Options may appear in any order and may be before, after, or between file names. The order of file names is significant.

-a[hls] [=*file*] Enable listings.
Enable listing output from the assembler. The option "**-a**" requests high-level, assembly and symbols listing. "**-ah**" requests a high-level language listing, "**-al**" requests an output-program assembly listing, "**-as**" requests a symbol table listing. The letters after "**-a**" may be combined into one option, for example "**-als**". "**-a=*file***" sends the listing output to ***file***.

-EB Generate big endian object code.

-EL Generate little endian object code. This is the default.

-I Include file search path.
Add a path to the list of directories that **st200as** searches for files specified in **.include** directives. "**-I**" may be used multiple times to include a variety of paths. The current working directory is always searched first; after that, **st200as** searches any "**-I**" directories in the same order as they were specified (left to right) on the command line.

-J Disable signed overflow warning.
Do not warn about signed overflow.

-L Include local labels.
Labels beginning with "**L**" (upper case only), "**$**", "**?**", or "**_?**" are called local labels. Normally these labels are not represented by symbols in the object file. This option instructs the assembler to create symbols representing local labels in the object file.

-mcore=st220|st231

Specifies whether the assembler should select the ST220, or the ST231 encoding table. The default, if no command line option is found, is ST220. It also encodes the corresponding information in the elf flags of the object file unless a **.assume st2*xx*** directive is found, in which case the elf flags will be the ones specified by the **.assume st2*xx*** directive.

-o *objectfile* Name the object file.
The default output file created by **st200as** is **a.out**. The output file name can be used with this option.

-R Join text and data sections.
Instructs **st200as** to merge the data section with the text section in the object file.

--statistics Display assembly statistics.
Use "**--statistics**" to display statistics about the resources used by **st200as**, such as the total execution time taken for assembly (in CPU seconds).

-v | --verbose Announce version.
To determine which version of **st200as** is executed, include the option "**-v**" on the command line.

-W Suppress warnings.
By default all warnings are printed to the standard error file. This option, disables the printing of warning messages.

-Z Generate object file in spite of errors.
After an error message, **st200as** normally produces no output. With this option a output file is generated (with possibly erroneous data).

11.3.1.2 Arguments

Except for "**--**", any command line argument that begins with a hyphen is an option. Each option changes the behavior of **st200as**. No option changes the way another option works. An option is a "**-**" followed by one or more letters; the case of the letters is important.

Some options expect exactly one parameter to follow them. The parameter may either immediately follow the option letter or it may be the next command line argument.

In the following example the two command lines are equivalent:

```
st200as -o my-object-file.o mumble.s
st200as -omy-object-file.o mumble.s
```

11.3.1.3 Input and output files

The phrase source program, abbreviated to source, is used to describe the program input to one run of **st200as**. The source program may be contained in one or more files; how the source is partitioned into files does not change the meaning of the source. By convention, assembler source files have a "**.s**" suffix.

The source program is a concatenation of the text in all the files, in the order specified.

Two hyphens "**--**" name the standard input explicitly as one of the files for **st200as** to assemble.

Each run of **st200as** produces an output file, the object file. If the source file is empty, **st200as** produces a small empty object file. The default output file name is **a.out**, but it may be renamed using the **-o** option.

11.3.2 Error and warning messages

st200as may write warnings and error messages to the standard error file (usually the terminal). Warnings report assumptions made so that **st200as** can continue assembling a flawed program; errors report grave problems that prevents the source program from being assembled. Fatal error messages cause assembly to stop immediately.

Warning and error messages have the format: (where ***nnn*** is a line number).

```
file_name:nnn:Message text
```

Revision history

Version	Date	Comments
H	June ‘05	Revised to incoprporate the R4.1 Toolset release.
G	November ‘04	Revised to incorporate final comments for the R4.0 Toolset release.
F	September ‘04	Revised to include IPA, C++ and to reflect the R4.0 Toolset release.
E	June ‘04	Revised to include ST231 and to reflect the ST200 R3.2 Toolset release.
D	January ‘04	Minor update to clear review comments.
C	October ‘03	Revised to reflect ST200 R3.1 Toolset release.
B	June ‘03	Format changes to conform to style guide.
A	May ‘03	First complete version, submitted to ADCS

Index

Symbols

comment character 176
##args 104
#pragma 111
  defaultinline 48, 64
  frequency_hint 47, 57
  indent 48, 58
  inline_file 48, 64
  inline_function 48, 64
  inline_next 48, 63
  ivdep 47, 50
  loopdep 47, 51-52
  loopmod 47, 53
  loopseq 47, 56
  looptrip 47, 54
  noinline_file 48, 64
  noinline_function 48, 64
  noinline_next 48, 63
  pipeline 47, 55
  unroll 47, 49
  weak 48, 58
$ character 109
$PATH 6, 133
& unary operator 106
&& unary operator 99
.comment 58
/* and */ comment characters 176
// characters 109
__alignof__ 110
__asm__ 119
__attribute__ 39
__BARE_BOARD__ 26
__BIG_ENDIAN__ 25
__builtin__divuw 146
__builtin__divw 146
__builtin__moduw 146
__builtin__modw 146
__builtin__prefetch 75
__builtin_classify_type 117
__builtin_constant_p 39, 116
__builtin_expect 39, 116
__builtin_prefetch 79, 117
__builtin_return_address 116
__byte__ 116
__cplusplus 26, 37
__DEPRECATED 37
__EXCEPTIONS 37, 44
__func__ 29
__FUNCTION__ 110
__GLIBCPP__ 43
__GLIBCXX__ 43
__gnu_cxx 41

__GNUC__ 24, 36, 98
__GNUC_MINOR__ 24, 36
__GNUC_PATCHLEVEL__ 36
__GNUG__ 37
__GXX_ABI_VERSION 37
__INLINE_INTRINSICS 26
__LITTLE_ENDIAN__ 25
__open64__ 24
__OPTIMIZE__ 26
__OS21_BOARD__ 26
__pointer__ 116
__PRETTY_FUNCTION__ 110
__restrict 13
__restrict__ 13
__ST200 25
__ST200__ 25
__st200__ 25
__ST200CC__ 24
__ST200CC_DATE__ 25
__ST200CC_MINOR__ 24
__ST200CC_PATCHLEVEL__ 24
__ST200CC_VERSION__ 25
__st200mul32 145
__st200mul64h 145
__st200mul64hu 145
__st200mulfrac 145
__ST220__ 25
__st220__ 25, 131
__ST231__ 25
__st231__ 25
__STDC_HOSTED__ 26
__STDC_VERSION__ 26
__STRICT_ANSI__ 25, 37
__VA_ARGS__ 104
__word__ 116
_GLIBCPP_VERSION 43
_GLIBCXX_VERSION 43
_LANGUAGE_ASSEMBLY 25
_LANGUAGE_C 25
_LANGUAGE_C_PLUS_PLUS 37
_Wmultichar 14

A

a.elf 6-8
ABI 190
abort 114
addcg 122
aggregate
  initializing 108
alias 38, 111-112
aliasing 71-75
align 38, 110
aligned attribute 111-112
alignof 38
ANSI 2, 13
-ansi 13, 109
array
  initializing 108
  variable length 103
  zero length 102
asm. See GNU ASM.
assembler. See st200as.
assembly language 1

B

Backus-Naur Form xv
bare machine 7, 21
binopt 21, 151, 166
BNF. See Backus-naur Form.
board.x 22
bootboard.o 22
bootcore.o 22
bootsoc.o 22

-Bsymbolic 91
bugs 147-149

C

-C 12
C 3
-c 11
C++ 3, 107, 109
C++ See also st200c++
C89 2, 13-14
C99 2, 13, 27-29, 97, 101, 104-108
  restrict keyword 13
clobber list 119
codereorder
  passing options to 21
comment
  assembler 176
  section 58
compiler. See st200cc.
complex.h 27
constant
  assembler 177
constructor 38, 113
  expressions 107
core 22, 25, 190, 193
core.x 22
cpp 98, 105, 111
  passing option to 21
  vararg macro 104
CVG dump 170
Cygwin 3

D

davinci 170
-dD 12
-Ddef 12
dead code elimination 151
debug information 2, 16, 152
default visibility 114
destructor 38, 113
-dM 12
-dN 12
documentation suite
  notation xv
-dumpversion 11
DWARF2 2, 16
dynamic_cast 43

E

-E 11-12
-EB 22, 25
-EL 22, 25
ELF 2, 97
endianness 2, 5, 22, 25
end-of-file 179
enum 110
escape character 110
etsitom.h 130
etsitost220.h 130
exit 113-114

F

-f[no-]access-control 34
-f[no-]check-new 34
-f[no-]dismissible-load 19
-f[no-]elide-constructors 35
-f[no-]exceptions 34, 37
-f[no]-for-scope 35
-f[no]-gnu-keywords 35
-f[no-]implicit-templates 34
-f[no]-permissive 35
-f[no-]rtti 34
-f[no]signed-bitfields 18
-f[no]strict-aliasing 17, 71-75
-f[no]unroll-loops 17

-f[no-]unsigned-bitfields 18
-f[no-]verbose-asm 19
-f[no-]zero-initialized-in-bss 19
-fb 17
-fb_create 17
-fcheck-new 44
-fdiagnostics-show-location 35
fenv.h 28
-ffixed-reg 18
-finstrument-functions 16, 81-83
float.h 28
floating-point 101
-fmessage-length 35
-fno-exceptions 31, 44
-fno-rtti 43
format 115
format_arg 115
-fpreprocessed 12
-fsigned-char 18
-ftemplate-depth 35
function
  calls 81-83
  prototype 109
-funsigned-char 18
-fvisibility 89-95
-fzero-initialized-in-bss 19

G

-g 16, 80, 88
gbd. See st200gbd.
GNU 1, 13, 97, 105
  ASM 119-122
  asm 39
  C++ See st200c++
  gas 175, 187
  style labeled elements in initializers 108
GNU89 13
GNU99 13
gprof. See st200gprof

H

-H 12
-help 11
hidden visibility 114
host 26

I

ICache
  optimization 151-174
--icache-algo 157, 159
--icache-mapping 156, 161
--icache-opt 156, 161
icacheopt 166-173
  options 167-171
--icache-profile 158, 161
--icache-static 157, 161
-Idirectory 23
IEEE-754 178
imodels.c 123, 130
imodels.h 130
imodels.o 131
initialize 108
-INLINE 60-61
  must 60
-inline 61
inline keyword 60
inlining 59-63
  pragmas 63-68
internal visibility 115
interprocedural analysis 86
intrinsics 123-145
  #include files 130
  absolute distance 136
  absolute value 136
  add 135

bit and byte ordering 144
bit manipulation 140
clamp 140
divide 138
extract 140
high precision fractional arithmetic 143
insert 140
integer 32-bit 142
left shift 137
long long 142
maximum 136
minimum 135
miscellaneous 139
modulus 138
multiply 137
multiply high 139
multiply special 138
negate 135
population count 144-146
right shift 137
round 140
subtract 135
switching model and ST220 names 131
viterbi 141
inttypes.h 28
-ipa 86
-IPA:aggr_cprop 87
-IPA:cgi 87
-IPA:cprop 87
-IPA:depth 87
-IPA:dfe 87
-IPA:dve 87
-IPA:forcedepth 87
-IPA:inline 87
-IPA:keeplight 87
-IPA:maxdepth 88
-IPA:multi_clone 88
-IPA:node_bloat 88
-IPA:plimit 88
-IPA:space 88
-IPA:specfile 88
ISO
1999 101, 104-109
iso646.h 27
ISO9899
1990 13
199409 13
1999 13

K

-keep 11

L

-l library 23
label
assembler 179
-Ldirectory 23
libboard.a 22
libcore.a 22
libgcc.a 134
libsoc.a 22
link register 184-185
linker. See st200ld.
Linux 3, 97
load application. See st200run
long long 101
lvalues 100, 106

M

-M 12
-m[no]auto-prefetch 76
macro
definition 100
variable number of arguments 104

main() 113
malloc 39, 114
math.h 28
-mboard 22
-mcore 22, 25, 190, 193
-MG 12
-minstrument-calls 16, 81-83
-MM 12
mode 115
-mruntime 21
-msoc 22
mtost220.h 131
-mvisibility-decl 89-95

N

new 44
newline 179
  embedded 105
no_instrument_function 81-82
-no-gcc 24, 36
-noinline 61
noreturn 39, 97, 114
-nostdinc 23
-nostdinc++ 36
-nostdlib 23

O

-O 16, 26
-o 11
-O0 16, 148
-O1 16, 148
-O2 16, 152, 161
-O3 16
-OPT 26
  0limit 62
  alias 69
  cray_ivdep 50
  liberal_ivdep 50
  unroll_size 50
optimization
  aliasing 71-75
  asm 122
  call trace instrumentation 81-83
  data prefetching 75-80
  GNU attributes 114
  icache and dead code 151
  icacheopt options 167-172
  inlining 59-69
  interprocedural analysis 86
  profiling feedback 83-85
  st200cc ICache options 156-159
  st200cc options. See -O0 to -O3 and -Os
  symbol visibility 89-95
  techniques 59-86
OS 19, 190
-Os 16
OS21 21

P

-P 12
pack 38
packed 111-112
path 6, 133
-pedantic 13, 15, 98
-pedantic-error 13
pft 117
-pg 16
prefetching 75-80, 117
preprocessor. See cpp.
printf 115
profiling 80
profiling feedback 83-85
protected visibility 115

R

-r 161-162

restrict 13, 27
rtti 43
run application. See st200run
run-time
  environment
    selecting 21
  libraries
    ICache optimization 161

S

-S 11
scanf 115
section 38, 111
SIGABRT 44
silicon 190
sizeof 106
soc.x 22
st200ar 5
st200as 1, 4-5, 7, 175-194
  bundles 182
  directives 187-191
  expressions 181-182
  operands 183-184
  operations 185-187
  passing options to 21
  symbols 179
  syntax 176-191
st200c++ 5, 31
  command line 32-36
  exceptions 44
  file types 32
  GNU asm 39
  GNU language extensions 38
  invoking 32
  limitations 31, 46
  new 44
  predefined macros 36
  standard template libraries 39-43
st200cc 1, 4-5
  bugs 147-149
  command-line 2, 5
    examples 6, 9-10
    ICache optimization examples 153, 163
  front-end 21
  input files 10
  interfaces 4
  invoking 9
  output files 10
  profiling feedback 83-86
  reporting a bug 147-149
st200gdb 2, 4-6, 8
st200gprof 80
st200ld 1, 4-5, 7, 9, 32, 112
  boot and target files 22
  compile without link 11
  ICache optimization 151, 154, 156-157, 160, 163-165
  intrinsics 131
  passing options to 21
  relocatable file 161-162
  selecting libraries 23
st200run 4-7, 165
ST220 1-2
st220.h 130
st220tom.h 131
st220types.h 126
ST231 2
standard template libraries. See st200c++
statement
  assembler 179
static
  function 61
  keyword 60
-std 13, 25, 27, 37
-std=c++98 33
-std=gnu++98 33
stdbool.h 29
stdint.h 28
stdio.h 28-29

strfmon 115
strftime 115
structure
  initializing 108
SunOS 3
symbol
  assembler 177
  visibility 89-95

T

target 6-8
tgmath.h 27
throw 44
TracePlugin 154-155
-traditional 109
typedef 109
typeid 43
typeinfo 43
typeof 38, 100

U

unary operator 99, 106
union 109
used 113

V

-v 11
vararg 60, 104
--version 11
visibility
  attributes 114
VLA 103
void 99, 106
volatile 121

W

-w 13
-W<phase>,<arg> 21
-W[no-]abi 33
-W[no-]ctor-dtor-privacy 33
-W[no-]deprecated 34
-W[no-]effc++ 33
-W[no-]old-style-cast 33
-W[no]overloaded-virtual 34
-W[no]pmf-conversions 34
-W[no-]reorder 34
-Waggregate-return 15
-Wall 13, 148
-Wbad-function-cast 14
-Wcast-align 15
-Wcast-qual 15
wchar.h 28
-Wchar-subscripts 14
-Wconversion 15
weak 38, 60, 111-112
  pragma 58
-Werror 13
-Werror-implicit-function-declaration 14
-Wformat 14
whitespace 176-177, 179
-Wimplicit 14
-Wimplicit-function-declaration 14
-Wimplicit-int 14
Windows 3
-Wlong-long 15
-Wmissing-braces 14
-Wmissing-declarations 15
-Wmissing-noreturns 15
-Wmissing-prototypes 15-17
-Wnested-externs 15
-Wno-deprecated 37
-Wpacked 15
-Wpadded 15
-Wparentheses 14

-Wpointer-arith 14
-Wredundant-decls 15
-Wreturn-type 14
-Wshadow 14
-Wsign-compare 15
-Wstrict-prototypes 15
-Wswitch 14
-Wtrigraph 14
-Wunknown-pragmas 14
-Wunused 14
-Wwrite-strings 15

XYZ

XVCG dump 170
-Y<phase>,<path> 23